cedar - C++ implementation of efficiently-updatable double-array trie

developed by Naoki Yoshinaga at Yoshinaga Lab., IIS, University of Tokyo

Skip to [Features | Download | History | Usage | Performance | Tuning | References]

About

Cedar implements an updatable double-array trie [1,2,3], which offers **fast update/lookup for skewed queries in real-world data**, e.g., counting words in text or mining conjunctive features in a classifier. Its update and lookup speed is comparable to (hard-to-serialize) hash-based containers (`std::unordered_map`) or modern cache-conscious tries [3,4] and even faster when the queries are short and skewed (see performance comparison).

The cedar is meant for those who still underestimate a double-array trie as a mutable container. **The double-array tries are no longer slow to update. It's actually fast**.

If you make use of cedar for research or commercial purposes, the reference will be:

> N. Yoshinaga and M. Kitsuregawa. **A Self-adaptive Classifier for Efficient Text-stream Processing.** Proc. COLING 2014, pp. 1091--1102. 2014.

Features

- **Fast update for skewed/ordered queries**: Cedar can incrementally build a double-array trie from a given keyset in time that is less than the other double-array trie libraries. As a dynamic container, the ordered insertion is 3x faster than `std::unordered_map`, while the random insertion is comparable.
- **Fast lookup for skewed/ordered queries**: Cedar, double-array trie, offers 2-30x faster skewed lookup than `std::unordered_map`, while the random lookup is comparable.
- **Serialization (load and store)**: Cedar can immediately de/serialize the resulting trie from/into disk as immutable/mutable dictionary, thanks to its simple, pointer-free data structure based on two (or three) one-dimensional arrays.
- **Small working space**: Cedar, double-array trie, compactly stores a keyset that shares common prefixes, and it needs smaller working space (in building a trie) than other trie libraries that support serialization. If a static trie suffices and you mind the size of a resulting trie, I recommend darts-clone (same lookup speed, half size), or marisa-trie.
- **Native support for 4-byte record**: Cedar stores up-to four-byte record in type specified by the first template parameter. The second and third template parameters specify two exceptional values that indicate specific lookup failures; no value or no path. The default exceptional values for no value / path are -1 / -2 for int record and `NaN` values for float record. You can of course associate a five- or more byte record with a key by preparing a value array and store its index (int) in the trie.
- **Sorted keys**: The keys in a trie are alphabetically sorted as in `std::map` unless a user sets the third template parameter `SORT_KEYS` to false (slightly faster update).
- **Parameter to control space/time trade-off for binary keyset** [2]: Setting a larger value to the fifth template parameter lets cedar thoroughly seek for empty elements in a trie to reduce its size. The default value usually gives a good trade-off in building a trie with low branching factor or fan-out. Currently, a binary key with '\0' inside is not supported.
- **Three trie implementations:** a **(normal) trie**, a **reduced trie** [3] (compact size and faster look-up for short keys), a **minimal-prefix trie** (compact size for long keys). A reduced trie is enabled if you put `#define USE_REDUCED_TRIE 1` before `#include <cedar.h>`, while a minimal-prefix trie is enabled if you `#include <cedarpp.h>` instead of `cedar.h`.
- **Simple, short, portable code**: Ceder is implemented as a C++ header in ~600 lines, without depending on C++-specific headers such as standard template libraries.

License: GNU GPLv2, LGPLv2.1, and BSD; or e-mail me for other licenses you want.

Download & Setup

```
> wget http://www.tkl.iis.u-tokyo.ac.jp/~ynaga/cedar/cedar-latest.tar.gz
> tar zxvf cedar-latest.tar.gz
> cd cedar-YYYY-MM-DD
> configure
> make install
```

Requirements

- **OS:** 32/64-bit UNIX-compatible OS (tested on Linux / Mac OS X)
- **Compiler:** tested with GNU gcc (≥ 4.0) or clang (≥ 2.9)

ToDo

- Support compaction after erasing keys (`shrink_to_fit()`).
- Support the zero-length key (empty string).
- Support keys including '\0'.

History

- **June 24th 2014** (**xz**) **(development; minor bug/typo fixes and docs only)**:
    - Change License from GNU GPL/LGPL to **GNU GPL/LGPL and 2-clause BSD license**.
    - Check whether value type is supported during compilation.
    - Fix a bug in `set_array()` in `cedarpp.h` (thanks to GK)
- **April 25th 2014** (**xz**):
    - Add **SWIG-based Perl/Python/Ruby/Lua bindings** (I forgot to add in the previous version); advantages over built-in hash in the script languages are
        - **smaller working space**: cedar significantly reduces memory consumption to 1/4 or less, with comparable insertion/look-up speed,
        - **better functionality**: cedar provides higher-level APIs such as prefix or suffix retrieval, along with serialization, and
        - **sorted key access**: cedar allows users to retrieve keys in a sorted order.
        - note: the current bindings suffer from overheads in script languages (x2 to x10 slower insertion/look-up compared to native C++).
    - **x1.1-x1.4 speed up** in building a trie (esp. prefix trie for binary keyset).
    - Implement a callback function in replacing a node.
- **January 29th 2014** (**xz**):
    - Add libdatrie to the benchmark program.
    - Implement `begin()` and `next()` to realize iterater-based (sorted) key traversal (depth-first search).
    - Reimplement `predict()` and `dump()` using `begin()` and `next()`; faster retrieval in tries other than double-array prefix trie (`cedarpp.h`).
    - Fix a bug in executing `dump()` for a trie loaded from file.
    - Return error code in `erase()` (`-1` if absent).
- **November 14th 2013** (**xz**):
    - **x1.1-1.2 speeding up** in looking up keys, while **reducing trie size by 40-60%** for long ASCII keys, by implementing a **minimal-prefix trie** (`cedarpp.h`)
        - `configure --enable-prefix-trie` to compile `cedar/mkcedar` with `cedarpp.h`.
        - `#include <cedarpp.h>` if you want to use this minimal-prefix trie in your code.
        - `shrink_tail()` removes unused elements in TAIL (automatically evoked in saving a trie if you set the (additional) third parameter of `save()` to `true`).
        - Since node indices are now 64-bit to record position in the trie and the tail for `update()`, you may need to use `cedar::npos_t` instead of `size_t`.
    - **x1.1-1.5 speeding up** in looking up keys, while **reducing trie size by 10-20%** for short binary keys, by pruning record nodes from trie (say, **reduced trie** [3]).
        - `configure --enable-reduced-trie` to compile `cedar/mkcedar` with this optimization.
        - `#define USE_REDUCED_TRIE` prior to `#include <cedar.h>` if you want to use this reduced trie in your code.
        - support only `int` record r ($0 \le r < 2^{31} - 1$).

- **x1.1 speed up** in inserting keys, by fixing a bug in searching empty elements; the fourth template parameter `MAX_TRIAL` is now in default set to `1` (formerly, `8`).

- **October 4th 2013** (**xz**):
  - Remove `#include <stdint.h>` to improve portability.
  - **x1.1-1.2 speed up** in inserting keys by optimizing collision resolution and block traversal, while reducing extra spaces needed.
  - Resolve potential memory leak in reallocating arrays and tiny memory leak in saving a mutable trie.
  - Fix a bug in passing an empty string to some functions (e.g., `update()` had destroyed (BASE of the root of) the trie).
    - `update()` rejects an empty string.
    - `erase()` skips an empty string.
    - `exactMatchSearch()` returns `false` for an empty string.
    - `commonPrefixPredict()` returns `0` for an empty string.
- **April 23rd 2013** (**xz**):
  - Support saving/loading a mutable trie; it recovers extra data needed to `update()` and `predict()` from an immutable trie (`configure --enable-fast-load` will save/load the extra data to directly save/load a mutable trie.)
  - Add doar, CMPH, and Array Hash to the benchmark program.
- **February 22nd, 2013** (**xz**):
  - Reduce maximum resident set size by exact-fit memory allocation (enabled by default).
    - `#define USE_EXACT_FIT` prior to `#include <cedar.h>`.
  - Add more libraries to benchmark programs (treap, skiplist, scapegoat tree, AA tree, ternary search tree, etc.).
- **February 9th, 2013** (**xz**):
  - Reduce maximum resident set size by resorting to legacy `std::realloc()`.
  - Add `total_size()` that returns memory consumed by the double array.
  - Add benchmark programs used in performance comparison to increase the reproducibility/reliability of the experiments.
- **January 12th, 2013** (**xz**):
  - Add `commonPrefixPredict()` to enumerate keys from given prefix.
  - Sort keys in a trie unless the fourth template parameter `SORTED_KEYS` is false.
  - Adopt intuitive names for template parameters.
  - Fix a bug in adding a key (infinite loop when open blocks become empty while relocating blanchings).
- **January 5th, 2013** (**xz**):
  - initial release.

Usage

Cedar provides the following class template:

```
template <typename value_type,
          const int     NO_VALUE  = nan <value_type>::N1,
          const int     NO_PATH   = nan <value_type>::N2,
          const bool    ORDERED   = true,
          const int     MAX_TRIAL = 1,
          const size_t  NUM_TRACKING_NODES = 0>
class da;
```

This declares a double array trie with `value_type` as record type; `NO_VALUE` and `NO_PATH` are error codes returned by `exactMatchSearch()` or `traverse()` if search fails due to no path or no value, respectively. The keys in a trie is sorted if `ORDERED` is set to true, otherwise siblings are stored in an inserted order (slightly faster update). `MAX_TRIAL` is used to control space and time in online trie construction (useful when you want to build a trie from binary keys). If `NUM_TRACKING_NODES` is set to a positive integer value, the trie keeps node IDs stored in its public data member `tracking_node` to be valid, even if `update()` relocates the nodes to different locations.

The default `NO_VALUE/NO_PATH` values for `int` record are `-1` and `-2` (same as darts-variants), while the values for `float` record are `0x7f800001` and `0x7f800002` (taken from NaN values in IEEE 754 single-precision binary floating-point format); they can be referred to by `CEDAR_NO_VALUE` and `CEDAR_NO_PATH`, respectively.

NOTE: `value_type` must be in less than or equal to four (or precisely `sizeof (int)`) bytes. This does not mean you cannot associate a key with a value in five or more bytes (e.g., `int64_t`, `double` or user-defined `struct`). You can associate any record with a key by preparing a value array by yourself and store its index in (`int`) the trie (`cedar::da<int>`).

Cedar supports legacy trie APIs adopted in darts-clone, while providing new APIs for updating a trie.

```
value_type& update (const char* key, size_t len = 0, value_type val = value_type (0))
```

Insert `key` with length = `len` and value = `val`. If `len` is not given, `std::strlen()` is used to get the length of `key`. If `key` has been already present int the trie, `val` is **added to** the current value by using `operator+=`. When you want to override the value, omit `val` and write a value onto the reference to the value returned by the function.

```
int erase (const char* key, size_t len = 0, size_t from = 0)
```

Erase `key (suffix)` of length = `len` at `from` (root node in default) in the trie if exists. If `len` is not given, `std::strlen()` is used to get the length of `key`. `erase()` returns -1 if the trie does not include the given key. Currently, `erase()` does not try to refill the resulting empty elements with the tail elements for compaction.

```
template <typename T>
size_t commonPrefixPredict (const char* key, T* result, size_t result_len, size_t len = 0, size_t from = 0)
```

Predict suffixes following given `key` of length = `len` from a node at `from`, and stores at most `result_len` elements in `result`. `result` must be allocated with enough memory by a user. To recover keys, supply result in type `cedar::result_triple_type` (members are `value`, `length`, and `id`) and supply `id` and `length` to the following function, `suffix()`. The function returns the total number of suffixes (including those not stored in `result`).

```
template <typename T>
void dump (T* result, const size_t result_len)
```

Recover all the keys from the trie. Use `suffix()` to obtain actual key strings (this function works as `commonPrefixPredict()` from the root). To get all the results, `result` must be allocated with enough memory (`result_len = num_keys()`) by a user.

NOTE: The above two functions are implemented by the following two tree-traversal functions, `begin()` and `next()`, which enumerate the leaf nodes of a given tree by a pre-order walk; to predict one key by one, use these functions directly.

```
int begin (size_t& from, size_t& len)
```

Traverse a (sub)tree rooted by a node at `from` and return a value associated with the first (left-most) leaf node of the subtree. If the trie has no leaf node, it returns `CEDAR_NO_PATH`. Upon successful completion, `from` will point to the leaf node, while `len` will be the depth of the node. If you specify some internal node of a trie as `from` (in other words `from != 0`), remember to specify the depth of that node in the trie as `len`.

```
int next (size_t& from, size_t& len, const size_t root = 0)
```

Traverse a (sub)tree rooted by a node at `root` from a leaf node of depth `len` at `from` and return a value of the next (right) leaf node. If there is no leaf node at right-hand side of the subtree, it returns `CEDAR_NO_PATH`. Upon successful completion, `from` will point to the next leaf node, while `len` will be the depth of the node. This function is assumed to be called after calling `begin()` or `next()`.

```
void suffix (char* key, const size_t len, size_t to)
```

Recover a (sub)string `key` of length = `len` in a trie that reaches node `to`. `key` must be allocated with enough memory by a user (to store a terminal character, `len + 1` bytes are needed). Users may want to call some node-search function to obtain a valid node address and the (maximum) value of the length of the suffix.

```
void restore()
```

When you load an immutable double array, extra data needed to do `predict()`, `dump()` and `update()` are on-demand recovered when the function executed. This will incur some overhead (at the first execution). To avoid this, a user can explicitly run `restore()` just after loading the trie. This command is not defined when you `configure --enable-fast-load` since the configuration allows you to directly save/load a mutable double array.

There are three major updates to the legacy APIs.

- `build()` accepts unsorted keys.
- `exactMatchSearch()` returns `CEDAR_NO_VALUE` if search fails.
- `traverse()` returns `CEDAR_NO_VALUE` if the key is present as prefix in the trie (but no value is associated), while it returns `CEDAR_NO_PATH` if the key is not present even as prefix.
- Because `update()` can relocate the node id (`from`) obtained by `traverse()`, a user is advised to store the traversed node IDs in public data member `tracking_node` so that cedar can keep the IDs updated. The maximum number of elements in `tracking_node` should be specified as the sixth template parameter (0 is used as a sentinel).

Performance Comparison

Implementations

We compared cedar (2014-06-24) with the following *in-memory and mutable* containers that support sequential insertions.

- Judy Array 1.0.5: Judy trie SL [11]
- hat-trie: HAT-trie [12]
- array-hash Array Hash: (cache-conscious) hash table [13]
- hopscotch-map: Hopscotch hash [14]
- sparsepp: sparse hash table
- sparsehash 2.0.2: dense hash table
- std::unordered_map <const char*, int> (`gcc-7.1`): hash table

These containers do not support serialization. We also compared cedar with the trie implementations that support serializations; the softwares other than cedar implement only *static or immutable* data structures that do not support update.

- Darts 0.32: double-array trie
- Darts-clone 0.32g: directed acyclic word graph
- Darts-clone 0.32e5: Compacted double-array trie [15]
- tx-trie*: LOUDS (Level-Order Unary Degree Sequence) trie [16]
- ux-trie*: LOUDS double-trie
- marisa-trie*: LOUDS nested patricia trie [17]

The succinct tries (*) do not store records; it needs extra space to store records (sizeof (int) * #keys bytes) and additional time to lookup it.

Unless otherwise versions are stated, codes in the latest snapshots of git or svn repository were used.

Settings (updating now...)

Having a set of keys and a set of query strings, we first build a container that maps a key to a unique value, and then look up the queries to the container. The codes used to benchmark containers (bench.cc and bench_static.cc) have been updated from the ones included in the latest cedar package. The slow-to-build or slow-to-lookup comparison-based containers and immutable double arrays are excluded from the benchmark (the previous results are still available from here). The entire keys (and queries) are loaded into memory to exclude I/O overheads before insertion (and look-up). All the hash-based containers use `const char*` in stead of expensive `std::string` to miminize the memory footprint and to exclude any overheads, and adopt `CityHash64` from Google's cityhash in stead of default hash functions for a fair comparison. To compute the maximum memory footprint required in inserting the keys into the data structures, we subtracted the memory usage after loading keys or queries into memory from the maximum memory usage of the process monitored by run.cc (for mac) (or run_linux.cc).

Since we are interested in the lookup performance for practical situations where a small number of frequent keys are queried significantly more than a large number of rare ones (Zipfian distribution), the following datasets are used for experiments.

- **Text dataset** from Dr. Askitis's website (see Datasets):
  - Key: distinct_1
  - Query: skew1_1
- **Binary dataset** (conjunctive features (integers) extracted from pecco dataset and encoded by variable byte coding):
  - Key: keys3 (sorted): extracted and uniquified from `pecco-YYYY-MM-DD/test/kernel_m3`
  - Query: test3: extracted from `pecco-YYYY-MM-DD/test/tl.dev`

The statistics of keyset are summarized as follow (a terminal symbol included):

Keyset	Text	Binary
Size [bytes]	304,562,809	144,703,811
Size (prefix) [bytes]	263,848,821	144,417,733
Size (tail) [bytes]	51,932,403	570,117
# keys	28,772,169	26,194,354
Ave. length [bytes]	10.58	5.52
Ave. length (prefix) [bytes]	9.17	5.51
Ave. length (tail) [bytes]	1.80	0.02
# nodes in trie	78,252,617	26,725,019
# nodes in trie (prefix)	37,538,629	26,439,275

From these statistics, we can compute the minimum size of double-array trie (cedar). It requires 8 * #nodes for internal nodes plus 8 * #keys for additional terminals that store 4-byte records. Concretely speaking, cedar requires at least 816.53 MiB (78,252,617 * 8 + 28,772,169 * 8 = 856,198,288) for text keyset, while it requires at least 403.74 MiB (26,725,019 * 8 + 26,194,354 * 8 = 423,354,984 bytes) for binary keyset.

Similarly, if a container stores raw keys and values, it requires at least 400.21MiB (304,562,809 + 4 * 28,772,169 = 419,651,485 bytes) for text keyset while it requires 237.92MiB (144,703,811 + 4 * 26,194,354 = 249,481,227) for binary keyset. Remember that a pointer (or offset) variable is additionally needed to access variable-length keys, while the memory alignment (e.g., 16 bytes in Mac OS X) incur severe overheads for short-length keys (if they are memory-allocated individually). If naively adopting `std::string` in hash-based containers, its space overheads are substantial (cf. previous resuls using `std::string`)

The statistics of queryset are summarized as follow (a terminal symbol included):

Queryset	Text	Binary
Size [bytes]	1,079,467,570	915,918,218
# queries	177,999,203	216,784,450
# keys	612,219	16,839,036
Ave. length [bytes]	6.06	4.23
Ave. key count	290.74	12.87

For the in-memory containers we measured the insertion and the lookup time per key. For the (static) tries we measured time to build the trie from the keyset (including time needed to save the trie into a hard disk; we alphabetically sort the keys in advance since some (static) tries require the sorted keys as input) and the lookup time per key.

The experiments were conducted on Mac OS X 10.11 over Intel Core i7-3720QM 2.6Ghz CPU with 16GB main memory. The benchmark codes are compiled for each container with `gcc-7.1 -O2 -g -std=c++17`. The best of five consective trials has been reported for each container.

Text dataset

The following table lists the timings needed in inserting keys (**Insert**), looking up the keys (**Lookup**), and maximum resident set size (memory) occupied by the process in inserting a trie (**Space**). Click the tabel headings for sorting.

Software	Data Structure	Space [MiB]	Insert [ns/key]	Lookup [ns/key]
cedar	Double-array trie	1171.74	651.47	54.59
cedar `ORDERED=false`	Double-array trie	1173.05	638.62	53.98
cedar	Double-array reduced trie	961.65	656.25	56.78
cedar `ORDERED=false`	Double-array reduced trie	961.55	635.98	56.49
cedar	Double-array prefix trie	680.63	845.29	52.35

Software	Data Structure	Space [MiB]	Insert [ns/key]	Lookup [ns/key]
cedar `ORDERED=false`	Double-array prefix trie	678.29	817.85	53.95
Judy 1.0.5	Judy trie SL	928.95	764.22	188.75
hat-trie	HAT-trie	**570.00**	620.40	64.27
Array hash	Array Hash	1575.01	691.87	**37.85**
Hopscotch hash	Hopscotch Hash	1581.38	395.80	48.25
sparsepp	Hash table (sparse)	1104.29	686.62	67.74
sparsetable 2.0.2 (dense)	Hash table (dense)	1941.53	**372.65**	43.74
std::unordered_map	Hash table	1843.65	546.84	103.38
		log		log

The following table lists the timings needed in building a trie from sorted keyset (**Build**), looking up the keys (**Lookup**), maximum resident set size (memory) occupied by the process in inserting a trie (**Space**), and the size of trie saved in hard disk (**Size**).

Software	Data Structure	Space [MiB]	Size [MiB]	Build [ns/key]	Lookup [ns/key]
cedar	Double-array trie	1006.57	816.54	205.39	42.95
cedar `ORDERED=false`	Double-array trie	1006.57	816.54	**182.35**	42.23
cedar	Double-array reduced trie	781.30	642.38	188.00	44.58
cedar `ORDERED=false`	Double-array reduced trie	787.31	642.38	**178.22**	44.38
cedar	Double-array prefix trie	**489.49**	488.38	214.58	39.98
cedar `ORDERED=false`	Double-array prefix trie	**489.48**	488.35	204.29	40.96
Darts 0.32	Double-array trie	4305.09	858.93	2522.85	40.54
Darts-clone 0.32g	Directed-acyclic word graph	2310.43	409.17	1359.94	**34.55**
Darts-clone 0.32e5	Compacted double-array trie	2778.06	**309.31**	1033.19	59.12
tx-trie 0.18	LOUDS trie	1345.68	*113.11	603.93	944.18
ux-trie 0.1.9	LOUDS two-trie	1729.32	*92.39	848.49	2059.98
marisa-trie 0.2.4	LOUDS nested patricia trie	2034.10	***87.27**	682.64	190.12
			log		log

* The succinct tries need additional space for record: 28,772,169 * sizeof (int) = 109.76 MiB (115,088,676 bytes).

Binary dataset

The following table lists the timings needed in inserting keys (**Insert**), looking up the keys (**Lookup**), and maximum resident set size (memory) occupied by the process in inserting a trie (**Space**).

Software	Data Structure	Space [MiB]	Insert [ns/key]	Lookup [ns/key]
cedar	Double-array trie	569.63	188.90	12.08
cedar `ORDERED=false`	Double-array trie	539.02	**155.40**	12.17
cedar	Double-array reduced trie	494.02	204.94	**8.15**
cedar `ORDERED=false`	Double-array reduced trie	495.35	172.47	**8.08**
cedar	Double-array prefix trie	537.62	224.46	9.72
cedar `ORDERED=false`	Double-array prefix trie	539.35	190.07	9.67
Judy 1.0.5	Judy trie SL	562.99	231.46	81.51
hat-trie	HAT-trie	**449.61**	223.02	50.09
Array hash	Array Hash	1395.09	642.51	75.27
Hopscotch hash	Hopscotch Hash	1568.71	395.42	61.19
sparsepp	Hash table (sparse)	982.46	700.29	109.09
sparsetable 2.0.2 (dense)	Hash table (dense)	1924.07	370.93	67.52
std::unordered_map	Hash table	1685.14	542.04	212.21
		log		log

The following table lists the timings needed in building a trie from sorted keyset (**Build**), looking up the keys (**Lookup**), maximum resident set size (memory) occupied by the process in inserting a trie (**Space**), and the size of trie saved in hard disk (**Size**).

Software	Data Structure	Space [MiB]	Size [MiB]	Build [ns/key]	Lookup [ns/key]
cedar	Double-array trie	486.53	403.75	190.48	8.42
cedar `ORDERED=false`	Double-array trie	486.73	403.75	**162.69**	8.66
cedar	Double-array reduced trie	**441.57**	368.14	206.56	**7.60**
cedar `ORDERED=false`	Double-array reduced trie	**443.65**	369.80	173.49	**7.31**
cedar	Double-array prefix trie	514.73	484.64	232.12	8.16
cedar `ORDERED=false`	Double-array prefix trie	516.20	486.42	193.07	8.22
Darts 0.32	Double-array trie	1573.27	406.03	338.32	7.70
Darts-clone 0.32g	Directed-acyclic word graph	1618.92	**201.91**	657.87	8.32
Darts-clone 0.32e5	Compacted double-array trie	2525.98	272.07	1180.55	15.43
tx-trie 0.18	LOUDS trie	1228.44	***38.63**	372.62	298.06
ux-trie 0.1.9	LOUDS two-trie	1152.00	***38.23**	322.44	1036.02
marisa-trie 0.2.4	LOUDS nested patricia trie	1438.70	*41.62	261.31	430.24
		log			log

* The succinct tries need additional space for record: 26,194,354 * sizeof (int) = 99.92 MiB (104,777,416 bytes).

Conclusions: Overall, the cedar variants are comparable to the cache-conscious or optimized hash containers in terms of speed to update. The cedar variants are also comparable to those containers in terms of speed to lookup on the text datasets (keys with low branching factors and longer tails) and are significantly faster on the binary datasets (keys with with high branching factors). Although the cedar variants require more space to de/serialize the resulting tries than the other immutable double-array or succinct tries, they require significantly smaller spaces in building a trie. **The double-array tries are no longer slow to update. It's actually fast.**

Disclaimer

We do not guarantee that the implemented algorithms other than those proposed by us are patent-free; we regarded them to be patent-free simply because their implementations are available as (existing) open-source softwares (otherwise a simple patent lookup). Please be careful when you use this software for commercial use.

How to pronounce `cedar'

The library is named after cedar, the most popular kind of tree in Japan; so just pronounce the same as cedar in English.

Acknowledgments

This software cannot be implemented without help of Dr. Yata, guru of double-array / succinct tries. The developer deeply appreciates his guidance on the implementation of the mechanism that controls space and time [2].

References

1. J. Aoe. **An efficient digital search algorithm by using a double-array structure**. IEEE Transactions on Software Engineering 15(9):1066--1077. 1989.
2. S. Yata and M. Tamura and K. Morita and M. Fuketa and J. Aoe. **Sequential Insertions and Performance Evaluations for Double-arrays**. Proc. the 71st National Convention of IPSJ, pp. 1263--1264. 2009.
3. N. Yoshinaga and M. Kitsuregawa. **A Self-addaptive Classifier for Efficient Text-stream Processing**. Proc. COLING 2014, pp. 1091--1102. 2014.
4. D. R. Morrison. **PATRICIA -- practical algorithm to retrieve information coded in alphanumeric**. Journal of the ACM. 15(4): 514--534. 1968.
5. D. Sleator and R. Tarjan. **Self-adjusting binary search trees**. Journal of the ACM, Vol 32(3), pp 652--686. July 1985. See demo in A demonstration of top-down splaying
6. C. R Aragon, R. Seidel. **Randomized Search Trees**. Proc. of FOCS, pp. 540--545. 1989.
7. W. Pugh. **Skip lists: a probabilistic alternative to balanced trees**. Communications of the ACM 33: 668--676. 1990.
8. I. Galperin and R. L. Rivest. **Scapegoat trees**. Proc. of SODA, pp. 165--174. 1993.
9. A. Andersson. **Balanced search trees made simple**. Proc. Workshop on Algorithms and Data Structures, pp 60--71. 1993.
10. J. L. Bentley and R. Sedgewick. **Fast Algorithms for Sorting and Searching Strings**. Proc. of SODA. pp. 360--369. 1997.

11. D. Baskins. **A 10-minute description of how Judy arrays work and why they are so fast**. http://judy.sourceforge.net/. 2004.
12. N. Askitis and R. Sinha. **HAT-trie: a cache-conscious trie-based data structure for strings**. Proc. of the 30th Australasian conference on Computer science. Vol 62, pp. 97--105. 2007.
13. N. Askitis and J. Zobel. **Cache-conscious Collision Resolution in String Hash Tables**. Proc. of SPIRE. pp. 91--102. 2005.
14. M. Herlihy and N. Shavit and M. Tzafrir. **Hopscotch Hashing**. Proc. of DISC. pp. 350--364. 2008.
15. S. Yata. **A compact static double-array keeping character codes**. Information Processing & Management, 43(1), pp. 237--247. 2007
16. G. Jacobson. **Space-efficient static trees and graphs**. Proc. SFCS, pp. 549--554. 1989.
17. S. Yata. **Dictionary Compression by Nesting Prefix/Patricia Tries**. Proc. JNLP, pp. 576--578. 2011.