Ordered hash map: search tree optimized by a hash table

Petar Ivanov, Valentina Dyankova, Biserka Yovcheva

***Abstract*—**An integration of a hash table and a balanced search tree is proposed. The new data structure, called *ordered hash map*, supports logarithmic-time inserts and constant-time finds and erases. An open source STL-style C++ implementations are developed which make the existing applications extremely easy to get optimized by simply changing the underlaying ordered class. Our *ordered hash map* is qualitatively and quantitatively compared to the heavily used *map* and *unordered_map* STL classes.

***Keywords*—**Algorithm complexity, Binary search tree, Hash.

I. Introduction

Information technologies nowadays demand not only on storing big data but also logically organizing the information enabling fast access. Sorting, or ordering, is one of the most heavily used ways of organizing information which allows search operation to be performed fast – typically in logarithmic-time of the number of stored elements. On the other hand hashing is a technique which allows constant-time searches but does not maintain the elements ordered. Our aim is to join the ordered and hashed approaches to obtain a data structure providing constant-time search operations while keeping the elements ordered for eventual traversal. Such a structure can be used for optimizing the execution time of existing applications performing many search queries in a dynamically ordered structure.

Sorting could be done by comparison the sorted elements or by other means (radix sort, counting sort, etc.). There are approaches of ordering a hash table [1] for faster search queries but without a fast traversal over the ordered elements. This paper considers the case of sorting by comparison.

II. The data structure

A. Preliminary definitions

We will call a *key* an object of a specific *key type* used as an argument of a mapping. The keys of a set are called *comparable* iff a total order relation on the set is defined. The keys of a set is called *hashable* iff every key can be hashed. A *data* is any object of a specific *data type* (also called *mapped type*) which is mapped by a key.

A *self-balancing binary search tree* [2], or simply a *tree*, is a binary search tree that automatically keeps its height small (for example: AVL tree, Red-black tree, etc.). A tree can be used for mapping comparable keys to data.

A *hash map* (or *hash table*) [3] is a data structure that can map hashable keys to data.

B. Ordered hash data structure

The introduced data structure *ordered hash* consists of a tree *T* which maps keys to data and a hash map *H* which maps keys to address pointers of elements of *T*. The main reason of integrating a hash table to a search tree is to speed-up the find operation while maintaining dynamically the order of the elements in the tree.

The main operations on the introduced ordered hash:

1) ***insert(key, value)***:
   - inserts the *(key, value)* pair into *T* as an element pointed by some pointer *p*.
   - inserts the *(key, p)* pair into *H*.
2) ***find(key)***:
   - searches for *key* in *H*.
   - returns a pointer to *key* in *T* if *key* is found in *H*.
3) ***erase(key)***:
   - erases *key* from *H*.
   - erases *key* from *T*.
4) ***next(tree pointer) / prev(tree pointer)***:
   - returns the *next*/*prev* pointer of the given *tree pointer* provided by the tree structure.

C. Implementation

The proposed C++ class *ordered_hash_map* is fully STL-consistent [4]. It has the same interface as STL *map* so as to be easily interchanged in existing projects. The class *ordered_hash_map* is templated by *key_type* and *data_type*. The type *ordered_t* of the tree *T* is STL *map<key_type, data_type>* and the type of the hash table *H* is the STL *unordered_map<key_type, ordered_t::iterator>* (note that the map specification guarantees that elements do not change their address).

The overhead memory usage of this implementation against map is *O(N*sizeof(key))* because *H* also contains the keys. The overhead is further lowered to *O(N)* by calculating the hash function on a temporal key.

The described *ordered_hash_map* class is templated by *Key*, *Data*, *Compare*, *HashFcn*, *EqualKey*, *MapAlloc* and *HashAlloc*. This class can be naturally extended to the analogous *ordered_hash_multimap*, *ordered_hash_set* and *ordered_hash_multiset* classes.

D. Complexity analysis

All the operations on STL *map* are available on *ordered_hash_map* with the same time and memory complexities except for the fast *find* operation which runs in *O(1)*. Note that if the number of elements *N* to be inserted is not known ahead, the complexities of insertion and erasion in hash tables are only amortized *O(1)* over multiple calls because of hash table resizing [5].

The insertion into *ordered_hash_map* consists of insertion into *T* for *O(logN)* and insertion into the *H* for *O(1)*.

The transitions to the next and to the previous element in the

All the authors are from the Computer Science department of Shumen University, 115 Universitetska str., 9700 Shumen, Bulgaria.

Petar Ivanov (corresponding author) – peter.ivanov89@gmail.com.
Valentina Dyankova – valentina.dyankova@gmail.com
Biserka Yovcheva – bissy_y@yahoo.com

defined order are done by traversing *T*. The maximum edge-distance between sequential elements is *O(logN)* but the average distance is *O(1)* amortized over iterating the whole tree (as every edge is traversed exactly twice – downwards and upwards).

	map	ordered hash map	ordered hash map*	unordered map
insert	logN	logN	logN*	**1**
find	logN	**1**	**1**	**1**
erase	logN	logN	**1**	**1**
next/prev traversal	**logN**	**logN**	**logN***	n/a or N
ordered traversal	**N**	**N**	**N***	n/a or NlogN

E. Erase optimization

Instead of erasing an element from both *H* and *T*, it can be only erased from *H*. Searching for an element will be considered unsuccessful iff the element is not found in *H*. This routine optimizes the erase execution time to *O(1)*. Some additional work (not increasing complexity) is to be done when traversing the tree by the *prev*/*next* operations: every element has to be checked on whether it was not erased from *H*. This is done by using *filter iterators* from Boost library [6].

Another implication of not physically erasing elements is that the iteration time is no more dependent on the number of elements *N* logically contained in the structure but on the number of elements *N** inserted ever before. *Ordered_hash_map* with the erase optimization is referred followed by a star – *ordered_hash_map**.

F. Next/prev optimization

It is also possible to lower the complexity of the transitions to *O(1)* by maintaining two additional pointers (for the next and the previous elements) for every element in *T*. This modification increases the time constant of the implementation while not lowering the expected number of opearations in a next/prev transition. This optimization is not implemented in *ordered_hash_map* as well as in the STL *map* containter.

III. Experiments

A. Artificial data

One million uniform random string containing 100 latin latters each are mapped to random integers. All strings are sequentially inserted to a data structure, then all the strings are searched against the full data structure, then all the strings are traversed in the sorted manner, and finally all the strings are sequentially erased. The table below compares the total execution times for multiple operations execution on each of the four data structures: STL *map*, *ordered_hash_map*, *ordered_hash_map** and STL *unordered_map* (hash table). The best execution times are highlighted. As expected, *unordered_map* inserts are much faster than inserting into trees because of the constant complexity. The slight difference of the find operation executions for *ordered_hash_map* and *ordered_hash_map** is due to the construction of a filter iterator. The slower *ordered_hash_map** traversal is due to the additional filter iterators.

	map	ordered hash map	ordered hash map*	unordered map
insert	2.68	3.60	3.63	**0.83**
find	2.67	**0.51**	**0.63**	**0.49**
erase	3.32	4.05	**0.94**	**0.91**
next/prev traversal	**0.21**	**0.22**	0.94	n/a

IV. Applications

Ordered hashes can be used instead of STL maps/sets for all kinds of hashable elements. Giving a hash function is not needed if STL containers are used as elements.

Sensitive hashing [7] and some other kinds of locally sensitive hashes are of a particular interest because of their inherited presumption for sorted hash values.

V. Conclusion

We hope that the presented data structure will be useful in practice because of its standard interface and its better complexity and find execution speed compared to the most commonly used alternatives. The future work includes testing the data structure on more real test cases.

All described C++ classes and detailed experimental results are available with open source licensing at https://github.com/petar-ivanov/ordered-hash.

VI. Funding

This paper is supported by the Project BG051PO001-3.3.06-0003 “Building and steady development of PhD students, post-PhD and young scientists in the areas of the natural, technical and mathematical sciences”.

References

[1] O. Amble and D. E. Knuth, Ordered Hash Tables, The Computer Jurnal (Oxford, 1974), Volume 17(2).

[2] D.Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition, Addison-Wesley, 1998, Section 6.2.3: Balanced Trees, pp.458—481.

[3] T. Cormen, C. Leiserson, R. Rivest, C. Stein, Introduction to Algorithms (2nd ed.), MIT Press and McGraw-Hill, 2001, pp.221—252.

[4] SGI Standard Template Library (STL) Programmer's Guide, Available: https://www.sgi.com/tech/stl/

[5] C. Leiserson, Amortized Algorithms, Table Doubling, Potential Method, Lecture 13, course MIT, Introduction to Algorithms – Fall 2005.

[6] D. Abrahams, J.Siek, T. Witt, Boost Filter iterators, Available: http://www.boost.org/doc/libs/1_55_0/libs/iterator/doc/filter_iterator.html

[7] C. Sadowski, Greg Levin, SimHash: Hash-based Similarity Detection