Hinted Dictionaries: Efficient Functional Ordered Sets and Maps

Amir Shaikhha
University of Edinburgh, UK

Mahdi Ghorbani
University of Edinburgh, UK

Hesam Shahrokhi
University of Edinburgh, UK

Abstract

Sets and maps are two essential collection types for programming used widely in data analytics [4]. The underlying implementation for both are normally based on 1) hash tables or 2) ordered data structures. The former provides (average-case) constant-time lookup, insertion, and deletion operations, while the latter performs these operations in a logarithmic time. The trade-off between these two approaches has been heavily investigated in systems communities [3].

An important class of operations are those dealing with two collection types, such as set-set-union or the merge of two maps. One of the main advantages of hash-based implementations is a straightforward implementation for such operations with a linear computational complexity. However, naïvely using ordered dictionaries results in an implementation with a computational complexity of $O(n \log(n))$.

Motivating Example. The following C++ code computes the intersection of two sets, implemented by `std::unordered_set`, a hash-table-based set:

```
std::unordered_set<K> result;
for(auto& e : set1) {
  if(set2.count(e))
       result.emplace(e);
}
```

However, the same fact is not true for ordered data structures; changing the dictionary type to `std::set`, an ordered implementation, results in a program with $O(n \log(n))$ computational complexity. This is because both the `count` (lookup) and `emplace` (insertion) methods have logarithmic computational complexity. As a partial remedy, the standard library of C++ provides an alternative insertion method that can take linear time, if used appropriately. The `emplace_hint` method takes a hint for the position that the element will be inserted. If the hint correctly specifies the insertion point, the computational complexity will be amortized to constant time.[1]

```
std::set<K> result;
auto hint = result.begin();
for(auto& e : set1) {
  if(set2.count(e))
       hint = result.emplace_hint(hint, e);
}
```

However, the above implementation still suffers from an $O(n \log(n))$ computational complexity, due to the logarithmic computational complexity of the lookup operation (`count`) of the second set. Thanks to the orderedness of the second set, one can observe that once an element is looked up, there is no longer any need to search its preceding elements at the next iterations. By leveraging this feature, we can provide a *hinted* lookup method with an amortized constant run-time.

[1] `https://www.cplusplus.com/reference/set/set/emplace_hint/`

36th European Conference on Object-Oriented Programming (ECOOP 2022).
Editors: Karim Ali and Jan Vitek; Article No. 33; pp. 33:1–33:3

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Hinted Data Structures. The following code, shows an alternative implementation for set intersection that uses such hinted lookup operations:

```
hinted_set<K> result;
hinted_set<K>::hint_t hint = result.begin();
for(auto& e : set1) {
  hinted_set<K>::hint_t hint2 = set2.seek(e);
  if(hint2.found)
    hint = result.insert_hint(hint, e);
  set2.after(hint2);
}
```

The above *hinted set* data-structure enables faster insertion and lookup by providing a cursor through a *hint object* (of type `hint_t`). The `seek` method returns the hint object `hint2` pointing to element `e`. Thanks to the invocation of `set2.after(hint2)`, the irrelevant elements of `set2` (which are smaller than `e`) are no longer considered in the next iterations. The expression `hint2.found` specifies if the element exists in `set2` or not. Finally, if an element exists in the second set (specified by `hint2.found`), it is inserted into its correct position using `insert_hint`.

The existing work on efficient ordered dictionaries can be divided into two categories. First, in the imperative world, there are C++ ordered dictionaries (e.g., `std::map`) with *limited* hinting capabilities only for insertion through `emplace_hint`, but not for deletion and lookup, as observed previously. Second, from the functional world, Adams' sets [1] provide efficient implementations for set-set operators. Functional languages such as Haskell have implemented ordered sets and maps based on them for more than twenty years [5]. Furthermore, it has been shown [2] that Adams' maps can be used to provide a parallel implementation for balanced trees such as AVL, Red-Black, Weight-Balanced, and Treaps. However, Adams' maps do not expose any hint-based operations to the programmer. At first glance, these two approaches seem completely irrelevant to each other.

The key contribution of this paper is hinted dictionaries, an ordered data structure that unifies the techniques from both imperative and functional worlds. The essential building block of hinted dictionaries are *hint objects*, that enable faster operations (than the traditional $O(\log n)$ complexity) by maintaining a pointer into the data structure. The underlying representation for hinted dictionaries can be sorted arrays, unbalanced trees, and balanced trees by sharing the same interface. In our running example, alternative data structures can be provided by simply changing the type signature of the hinted set from `hinted_set` to another implementation, without modifying anything else.

2012 ACM Subject Classification Software and its engineering → Functional languages; Theory of computation → Data structures design and analysis

Keywords and phrases Functional Collections, Ordered Dictionaries, Sparse Linear Algebra

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.33

Category Extended Abstract

Related Version *Full Version*: `https://arxiv.org/abs/2206.04380`

Acknowledgements The authors would like to thank Huawei for their support of the distributed data management and processing laboratory at the University of Edinburgh.

References

1 Stephen Adams. Efficient sets – a balancing act. *JFP*, 3(4):553–561, 1993.

2 Guy E Blelloch, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. In *SPAA'16*, pages 253–264, 2016.

3 Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *PVLDB*, 2(2):1378–1389, 2009.

4 Amir Shaikhha, Mathieu Huot, Jaclyn Smith, and Dan Olteanu. Functional collection programming with semi-ring dictionaries. *PACMPL*, 6(OOPSLA1):1–33, 2022.

5 Milan Straka. The performance of the haskell containers package. *ACM Sigplan Notices*, 45(11):13–24, 2010.