**Auhtor** : **Rai Ali Yar**

Department of computer science, FAST NU, Pakistan

**Author Email** : [raiali5030@gmail.com](mailto:raiali5030@gmail.com)

Research Paper: **Achieving O(1) Time Complexity for Insertion and Deletion in Dictionary Structures using Associative Arrays**

**Abstract:**

This research introduces a groundbreaking O(1) time complexity approach for associative arrays in C++, using a single line of code to optimize insertion, searching, deletion, and updating. Notably, the study addresses the intricate challenge of duplication handling during insertion, ensuring data integrity while maintaining constant-time efficiency. Beyond applications in databases and natural language processing, the findings contribute valuable insights into broader data structure landscapes. The succinct approach to duplication handling reflects a transformative leap in efficiency and scalability for dynamic datasets.

**1. Introduction:**

**We look at structures such as trees, graphs, linked list, queues and many more the one thing which is common in all that is whenever we do some operations code starts to iterate itself like getting in loops or going for a pointer connected to next next and many other ways. The point of dissicsion is that when code start more links joined to each other it becomes complex and difficult to apply operations. So our first approach is to make the code as less difficult as it could. When we map a information on paper our mind knows where we have written the following information which can be said as schema created and when we want to apply any operation on that information we directly access it by that schema that we maked a unique one which is distinct to other information in brain. So if we take schema as a index and put our unique element of data there in the index, then the information linked to it can be placed in front of the index. This lead us to an most easy and fastest way to access the data, then performing operations is going to be a simple task. This way of storing data is called associative arrays. In many languages it is available.**

**1.1 Background:**

Dictionary structures, vital in numerous applications, serve as key-value pairs, providing a fundamental means of organizing and retrieving data efficiently. They play a critical role in applications such as natural language processing, databases, and information retrieval. The effectiveness of dictionary structures lies in their ability to handle key operations—insertion, searching, deletion, and updating—with optimal time complexity.

In the realm of real-world applications, the efficiency of dictionary operations becomes paramount, especially when dealing with large datasets and dynamic updates. Traditional methods for managing these operations may fall short in meeting the demands of real-time processing and scalability. For example, in applications requiring frequent updates to dictionaries, like language translation services or dynamic databases, achieving optimal time complexity for these key operations is essential.

This research is motivated by the comprehensive optimization of dictionary operations in a language-agnostic manner. The primary objective is to explore a generic approach that ensures the efficient handling of insertion, searching, deletion, and updating operations with optimal time complexity, specifically targeting O(1) complexity for each operation. The research aims to provide a universal solution applicable across different programming languages, aligning with the increasing demands for real-time and scalable systems in diverse computing environments.

Efforts to streamline these fundamental operations in dictionary structures have profound implications, influencing the responsiveness of applications and overall user experience. The outcomes of this research are expected to contribute significantly to various domains, including enhancing the efficiency of language processing algorithms, optimizing database management systems, and improving information retrieval mechanisms.

As we delve into the methodology and results sections, a language-agnostic exploration of how these concepts are implemented and validated will be presented. This approach ensures the broad applicability and adaptability of the proposed techniques to various programming languages, making it a versatile solution for addressing the efficiency challenges of dictionary structures in their entirety.

**1.1.1 Insertion**

Insertion is a fundamental operation in dictionary structures, encompassing the addition and is often a critical factor in scenarios where data is dynamically changing. Efficient insertion ensures that new information seamlessly integrates into the dictionary structure without causing significant delays.

This research focuses on devising a generic approach that ensures optimal time complexity for insertion operations across various programming languages. Specifically, the study explores the use of associative arrays, exemplified by C++'s `unordered\_map`, as a vehicle for achieving O(1) time complexity for insertions.

**1.1.2 Searching:**

Searching within a dictionary is a core functionality that influences the speed and accuracy of information retrieval. Whether it's finding the meaning of a word in a language processing application or locating a specific record in a database, the efficiency of the search operation is paramount.

This research aims to extend the language-agnostic approach to searching within dictionary structures. By utilizing the chosen associative array structure, the study explores methods to achieve O(1) time complexity for search operations. Rapid access to values associated with a given key becomes crucial, contributing to the overall efficiency and effectiveness of dictionary-based applications.

**1.1.3 Deletion:**

Deletion operations involve the removal of key-value pairs from the dictionary. In scenarios where data is constantly changing or becoming obsolete, efficient deletion is crucial for maintaining the integrity and relevance of the dictionary structure.

The proposed research investigates language-agnostic techniques for achieving O(1) time complexity in deletion operations. By exploring methods within associative arrays that allow for swift removal of elements, the study aims to contribute to the optimization of dictionary structures in applications where data is frequently updated or removed.

**1.1.4 Updating:**

Updating involves modifying the value associated with a given key, ensuring that the dictionary reflects the most current information. In applications requiring real-time data updates, efficient updating is essential for maintaining accuracy.

The research delves into language-agnostic strategies for achieving O(1) time complexity in updating operations. By exploring techniques within associative arrays that facilitate quick updates, the study aims to enhance the efficiency of dictionary structures in applications with dynamic and evolving datasets.

As we proceed with the methodology and results sections, a detailed exploration of how these language-agnostic techniques are implemented and validated will be presented. The goal is to provide a comprehensive solution that optimizes dictionary operations across various programming languages, meeting the demands of real-time and scalable systems.

**1.3 Contribution:**

This work contributes to the field by leveraging C++'s **`unordered\_map**` for quick and scalable dictionary operations. The research explores the practical implications of achieving constant time complexity in real-world applications.

**3. Methodology:**

**3.1 Data Representation:**

To achieve a language-agnostic approach, the research focuses on utilizing associative arrays, a versatile data structure found in many programming languages. Associative arrays allow for the efficient storage and retrieval of key-value pairs, forming the backbone of our dictionary structure. The choice of associative arrays is motivated by their inherent O(1) time complexity for average-case insertion, searching, deletion, and updating operations.

**As the data could be in any form it has always a key element in it. The key element can be called unique value(key)** *fig1.1***. In the working done we place the unique key element in the indexes of the associative array. Then we have a type of hash table which has unique key as the indexes and the data linked to it is stored in a vector as we know a vector has a dynamic array. By our mean we can add data.**

**3.2 Insertion Operation:**

The insertion operation involves adding new key-value pairs to the dictionary. In the chosen language-agnostic approach, we leverage the native capabilities of associative arrays to achieve O(1) time complexity for insertion.

**To add data in this structure is so much simple just place the unique key in the index as in figure 1.1 shows. We can have associative arrays in many programming languages as fig 1.1,fig 1.2 and 1.3 demonstrates.**

Python Time complexity O(1)

def insert(dictionary, key, value):

dictionary[key]= value **//key is the unique element of data**

*fig 1.2*

Javascript Time complexity O(1)

function insest (dictionary,key,value){

dictionary[key]= value; **//key is the unique element of data**

}

*fig 1.3*

C++ Time complexity O(1)

void insert (string key){

dictionary[key]= value; **//key is the unique element of data**

}

*fig 1.4*

|  |  |
| --- | --- |
| Index(Unique key) | Data *(via node or vector etc)* |
| *Key 1* | *Vector 1* |
| *Key 2* | *Vector 2* |
| *Key 3* | *Vector 3* |
| .  .  . | **.**  **.**  **.** |
| *Key n* | *Vector n* |

*Fig 1.1*

**3.3 Searching Operation:**

Searching within the dictionary involves retrieving the value associated with a given key. The language-agnostic approach ensures O(1) time complexity for search operations.

**When we search an element it straight away checks the index having such value. If that value index is found it means that the element exist else not found. Shown above on fig 1.1 if key 2 is searched the following codes of fig 1.5-1.7 gives the best and fastest searching code as in O(1) the element is searched from the associative array.**

Python Time complexity O(1)

def search(dictionary, key):

return dictionary.get(key, None*)* //**if key is there then value exist**

*fig 1.5*

Javascript Time complexity O(1)

function search(dictionary, key) {

return dictionary[key]; //**if key is there then value exist**

}

*fig 1.6*

C++ Time complexity O(1)

bool search(string word) {

if (!dictionary[word].empty()) //**if key is there then value exist**

return true;

}

*fig 1.7*

**3.4 Deletion Operation:**

Deletion operations entail removing key-value pairs from the dictionary. The language-agnostic approach leverages the native capabilities of associative arrays to achieve O(1) time complexity for deletion.

**If a element is to chosen to delete the given element key is accessed and delete in one iteration, codes in fig 1.8-1.10 shows that an element is passed to the function. Firstly element searched is the element exist it would be deleted at most two time associative array is being accessed in the code. Clearly demonstrates time complexity of O(1).**

Python Time complexity O(1)

def delete(dictionary, key):

if key in dictionary:

del dictionary[key**] // key is directly accessed and deleted**

*fig 1.8*

Javascript Time complexity O(1)

function deleteEntry(dictionary, key) {

if (key in dictionary) {

delete dictionary[key]; **// key is directly accessed and deleted**

}

}

*fig 1.9*

C++ Time complexity O(1)

void deletion(const string& word) {

if (!dictionary[word].empty()) {

dictionary.erase(word); **// key is directly accessed and deleted**

}

}

*fig 1.10*

**3.5 Updating Operation:**

Updating involves modifying the value associated with a given key. The language-agnostic approach ensures O(1) time complexity for updating operations.

**As we want to update the values of key element we can easily do it by first checking, does such element exist whose values are to be updated then the value of that element is updated.**

Python Time complexity O(1)

def update(dictionary, key, new\_value):

if key in dictionary:

dictionary[key] = new\_value

Javascript Time complexity O(1)

function update(dictionary, key, newValue) {

if (key in dictionary) {

dictionary[key] = newValue;

}

}

C++ Time complexity O(1)

void update(string prevkey , string newkey) {

if (!dictionary[prevkey].empty()) {

dictionary[prevkey].emplace(newkey);

}

}

The implementation of these operations across different programming languages demonstrates the versatility and adaptability of the proposed language-agnostic approach. The next section will present experimental results and performance analysis to validate the efficiency of these operations.

**3.6 Data Loading:**

Data is loaded from the memory via any route all the data that is to be stored can be extracted is passed to the insetion function fig 1.2-1.4. Then, key element generates a place in the associative array then the data of that element is placed at the content of that index(having index as element).

|  |
| --- |
| Figure 2.1 Shows how steps in which data is loaded and inserted |

**4. Experimental Results:**

**4.1 Benchmarks:**

C++ is the language on which the main working was done at beginning. So in C++ we don’t have direct access to associative arrays, in order to use it we use built in library

#include <unordered\_map>

By using this library we get to access to a associative arrays, the structure we define can be made by our needs as the kind of data types.

We had used a large set of data of dictionary approximate 50,000 words and there meaning. Moreover, many words have duplication as having more than one meaning of a single word. The

Code we have elemented efficiently handles duplication.

First we have to define the data type and structure of the associative array the struture we have maded is shown in fig 4.1

**unordered\_map<string, vector<string>> dictionary;**

**Associative array**

|  |  |
| --- | --- |
| Index( Word Unique key) | Content |
| *Word 1* | *Meaning 1* |
| *Word 2* | *Meaning 2* |
| *Word 3* | *Meaning 3* |
| .  .  . | **.**  **.**  **.** |
| *Word n* | *Meaning n* |

*fig 4.1*

The thing which makes a bench mark is that duplicationsa are handled so well that it does not distrubs codes time complexity or iterations. As we are using vectors, it is known that vector is a dynamic array so when a word has another meaning it is just “pushback()” .

dictionary[word].push\_back(meaning);

This single line of code does all the insertion this same line handles the duplication this is a benchmark which clearly states that no other structure have a single line code for insertion and same at a time handling duplications.

**4.2 Performance Analysis:**

The performance of such a structure can only be lagged if there are any type of collisions if we see deeply into the structure it clearly represents a kind of hash table so when we talk about hashing first thing came in mind is collision which can in nano seconds destroy time complexity from O(1) O(N).

The hash key we need should be such in way unique that it never have a collision so in order to get it we have to get such set of elements of our data which never repeats itself. And in a dictionary we have the word itself as a unique key, if it repeats it means it is a synonym there we have another meaning for it, therefore no collision in data. So we never get a collsion so performance wise it never goes above then O(1).

**5. Advantages of this structure**

**5.1 Pros of using this structure:**

Beyond expectations this structure is great in it self as when time complexity is talked it get the best ,worst, average case of O(1). Then we had a second major and the most important thing in the going world is space and how much runtime ram and cache is having burden. The answer to it is that not a single byte of data is consumed otherthan the size of data entered.

If we disscuss cases of space complexity of other structures it is more than associative arrays. In trees or graphs the space complexity is more as we have pointers there address being hold in memory then link inside the link takes memory which at the end increases the space complexity .

5.2 Real-World Applications:

Highlighting potential applications of the proposed solution in real-world settings is important to show the real worth of this working as below all categories are in need of this structure

1. **Caching Systems:**

Associative arrays are commonly used in caching systems. The ability to quickly look up cached data based on a key (e.g., a URL or query) allows for fast retrieval of frequently accessed information.

1. **Databases:**

In-memory databases often use hash indexes or hash tables to achieve O(1) time complexity for search and retrieval operations. This is especially useful for scenarios where quick access to records based on keys is essential.

1. **Language Runtimes:**

Many programming languages use associative arrays (or similar structures like dictionaries or maps) for their internal symbol tables or variable lookup tables. This enables fast access to variables or functions during program execution.

1. **Web Servers and Load Balancers:**

Associative arrays are employed in web servers and load balancers to efficiently manage and lookup client sessions, routing information, or configuration settings based on keys.

1. **Cryptography:**

Some cryptographic algorithms rely on quick access to precomputed values or tables, where associative arrays can be used to store and retrieve these values efficiently.

1. **Compiler Symbol Tables:**

Compilers often use hash tables to manage symbol tables, allowing for efficient lookup of identifiers during the compilation process.

1. **Network Routing Tables:**

In networking, hash tables can be utilized in routing tables for fast lookups of destination addresses, facilitating efficient packet routing.

1. **Operating System File Systems:**

File systems may use hash tables or similar structures for fast access to file metadata, such as attributes or permissions, based on file names.

1. **Distributed Systems:**

In distributed systems, associative arrays can be employed for key-based data partitioning and quick access to distributed datasets, allowing for efficient data retrieval in a decentralized environment.

1. **Caching DNS Resolvers:**

DNS resolvers often use caches with associative arrays to store the results of previous DNS queries, enabling faster resolution of frequently requested domain names.

1. **Configuration Management:**

Systems that manage configurations and settings often use associative arrays to quickly retrieve configuration values based on keys.

**6. Limitations and Future Work:**

**6.1 Acknowledging Limitations:**

As we have seen enough bueaty of this structure, if we talk about the limitation it holds in such that when we want suggestions below while searching it will not be possible as this is not a trie tree structure so we don’t have proper set of defined values all data is scattered.

**6.2 Future Directions:**

In future working deep in the associative arrays in the built code of it there is in need of a less scattered data set is so far now we don’t have information about which element is where which makes some operations almost impossible to achieve on of them is suggestion box.

I hope in future I get proper access to the language source code I want that there is such implementation that data being added in array is sorted without sorting it. It hears impossible but I mean is we suppose we have 12403 elements in data when I enter a element it alphabetical letter is break down and checked if it exist if it is to where it’s letter is there till that point automatically memory of the new element is next to it in this way data will be stored in less then n iteration n means by number of letters in word.

**7. Conclusion:**

**7.1 Summary:**

In conclusion, this research has delved into the optimization of associative arrays to achieve O(1) time complexity for crucial operations like insertion, searching, deletion, and updating. The implementation, demonstrated through a C++ codebase utilizing `unordered\_map`, showcases the feasibility and applicability of the proposed techniques.

The investigation into achieving constant-time operations, even in dynamic datasets, holds significant implications for various domains. The ability to provide fast and efficient access to data based on keys enhances the performance of applications ranging from databases and caching systems to natural language processing and networking.

The findings contribute to the broader field of data structures, offering insights into scalable and responsive solutions for handling large-scale datasets. The adaptability of the proposed techniques to dynamic environments positions them as valuable assets for systems requiring real-time updates and efficient data retrieval.

As we look to the future, the research opens avenues for integration with emerging technologies, suggesting a continuous evolution of data structures in alignment with the changing landscape of computing. The potential applications extend beyond traditional paradigms, influencing algorithmic decision-making and shaping the efficiency of systems in diverse domains.

In summary, the research not only achieves its primary goal of O(1) time complexity in associative arrays but also sets the stage for advancements in data structures, offering a promising trajectory for the optimization of key operations in dynamic and large-scale computing environments.

**7.2 Implications:**

**Advancements in Data Structures:**

The research contributes to the broader field of data structures by providing insights into achieving O(1) time complexity for operations in associative arrays. This could inspire advancements in designing more efficient and scalable data structures for diverse applications.

**Performance Optimization in Key Applications:**

The achieved O(1) time complexity for insertion, searching, and other operations in associative arrays has profound implications for performance optimization. This can directly benefit key applications such as databases, caching systems, and language runtimes where fast access to data based on keys is paramount.

**Scalability in Large-Scale Systems:**

Large-scale systems, including web servers, distributed databases, and cloud services, can leverage the research findings to enhance scalability. The ability to achieve constant-time operations even as the dataset grows contributes to improved efficiency in handling vast amounts of data.

**Real-Time Processing in Natural Language Applications:**

The research findings have potential applications in natural language processing systems where real-time access to dictionary-like structures is crucial. This can impact applications ranging from language translation to chatbots, enabling more responsive and efficient processing of language data.

**Enhanced User Experience in Information Retrieval:**

Improved efficiency in dictionary structures translates to enhanced user experiences in information retrieval applications. Search engines, content recommendation systems, and knowledge bases can benefit from faster and more responsive retrieval of relevant information.

**Optimization in In-Memory Databases:**

In-memory databases rely on fast data access for optimal performance. The research has implications for the optimization of in-memory databases, providing a foundation for achieving O(1) time complexity in key operations, thereby enhancing overall database performance.

**Reduced Latency in Networking and Routing Systems:**

Networking and routing systems often require quick access to routing information based on destination addresses. The research findings can reduce latency in such systems, contributing to more responsive and efficient network routing.

**Adaptability in Dynamic Environments:**

The adaptability of the proposed techniques in dynamic datasets and real-time updates has implications for applications that operate in dynamic environments. Systems handling frequently changing data can benefit from the constant-time operations even in the face of dynamic updates.

**Potential for Integration with Emerging Technologies:**

The research opens avenues for integration with emerging technologies, such as non-volatile memory, quantum computing, and machine learning. This adaptability positions the findings at the forefront of innovations that may shape the future of efficient data storage and retrieval.

**Influence on Algorithmic Decision-Making:**

As data structures form the backbone of many algorithms, the research findings can influence algorithmic decision-making processes. Algorithms relying on associative arrays for fast access to key-based information may see performance improvements, impacting a range of applications.

**Acknowledgements:**

The author would like to express sincere gratitude to **Sir** **Muhammad Adeel** ( m.adeel@nu.edu.pk)for their invaluable guidance and support throughout the research process. Their expertise and mentorship played a pivotal role in shaping the outcomes of this study.

**8. References:**

* Books : Introduction to theory of computation by micheal sipser
* https://drive.google.com/file/d/1PqC30bPR2anlvm\_wI8ZphHK5IXlaE7ZR/view?usp=drive\_web&authuser=0
* Quoura : Alen guo for the knowledge and further recommendations
* Quoura : Sujeet qumar for real time applications
* <https://ieeexplore.ieee.org/document/10007469>
* [Research paper : Improving the time and space complexity of the WFA algorithm and generalizing its scoring by  JordanMEizenga ,  Benedict Paten](https://www.sciencegate.app/document/10.1101/2022.01.12.476087)