Data Structures & Algorithms (C++)

Dictionary ADT

1. The Dictionary ADT

Introduction

A Dictionary Abstract Data Type (ADT) is a data structure that stores data in **key–value pairs**, where each key is unique and is used to access its associated value.

It is similar to a map or associative array and is one of the most important ADTs in computer science.

Dictionaries are widely used in databases, search engines, compilers (symbol tables), and caching systems.

Definition

A Dictionary ADT supports operations such as:

- Insert(key, value)
- Remove(key)
- Find(key)
- Update(key)

Concept Example

Key → Value
"ID101" → "Ali"
"ID102" → "Sara"
"ID103" → "Khan"

Characteristics

- Keys are unique
- Values may repeat
- Fast retrieval using key
- Dynamic structure

Importance

- Forms basis of maps and hash tables
- Used in real-world applications like databases
- Enables fast searching and indexing

2. C++ Dictionary Implementation

Introduction

A dictionary in C++ can be implemented using arrays, linked lists, or hash tables.

The most efficient implementation uses hashing, but a simple version can be built using vectors or lists for educational understanding.

This implementation stores key-value pairs and performs linear search for simplicity.

C++ Implementation (Simple Dictionary)

```
#include <iostream>
#include <vector>
using namespace std;

template <typename K, typename V>
class Dictionary {
private:
   vector<pair<K, V>> data;

public:
   void insert(K key, V value) {
      for (auto &item : data) {
         if (item.first == key) {
            item.second = value; // update
            return;
         }
      }
      data.push_back({key, value});
   }

   bool find(K key, V &value) {
      for (auto &item : data) {
         if (item.first == key) {
            value = item.second;
            return true;
         }
      }
      return false;
   }

   void remove(K key) {
```

```
        for (auto it = data.begin(); it != data.end(); it++) {
            if (it->first == key) {
                data.erase(it);
                return;
            }
        }
    }

    void display() {
        for (auto &item : data) {
            cout << item.first << " -> " << item.second << endl;
        }
    }
};
```

Main Function Example

```
int main() {
    Dictionary<int, string> dict;

    dict.insert(1, "Ali");
    dict.insert(2, "Sara");
    dict.insert(3, "Omar");

    cout << "Dictionary contents:\n";
    dict.display();

    string value;
    if (dict.find(2, value))
        cout << "\nFound: " << value << endl;

    dict.remove(1);

    cout << "\nAfter deletion:\n";
    dict.display();

    return 0;
}
```

Complexity

Operation	Time Complexity
Insert	O(n)
Search	O(n)
Delete	O(n)

Advantages

- Simple to implement
- Flexible key-value storage
- Easy to understand for beginners

Disadvantages

- Slow for large datasets
- Linear search required
- Not optimized like hash tables or STL map

3. Implementations with Location-Aware Entries

Introduction

A location-aware dictionary stores not only key-value pairs but also tracks the **position (index or memory location)** of each entry.

This is useful in systems like compilers, file indexing, and databases where the location of data matters.

It enhances performance by allowing direct access or faster updates.

Concept

Key	Value	Location(Index)
A	100	0
B	200	1
C	300	2

Why Location Awareness?

- Faster updates
- Efficient deletion
- Tracking memory positions
- Used in symbol tables

C++ Implementation (Location-Aware Dictionary)

```
#include <iostream>
#include <vector>
using namespace std;

template <typename K, typename V>
```

```
class LocationAwareDictionary {
private:
   vector<pair<K, V>> data;

public:
   void insert(K key, V value) {
      data.push_back({key, value});
   }

   void showLocations() {
      for (int i = 0; i < data.size(); i++) {
         cout << "Key: " << data[i].first
              << " Value: " << data[i].second
              << " Location: " << i << endl;
      }
   }

   int getLocation(K key) {
      for (int i = 0; i < data.size(); i++) {
         if (data[i].first == key)
            return i;
      }
      return -1;
   }
};
```

Example Usage

```
int main() {
   LocationAwareDictionary<int, string> dict;

   dict.insert(1, "Ali");
   dict.insert(2, "Sara");
   dict.insert(3, "Khan");

   dict.showLocations();

   cout << "\nLocation of key 2: "
        << dict.getLocation(2) << endl;

   return 0;
}
```

Advantages

- Tracks memory/index positions
- Faster access in some systems
- Useful for compilers and databases

Disadvantages

- Extra storage overhead
- Not suitable for dynamic rehashing systems

4. Exercises (Exam / Practice Questions)

Theoretical Questions

1. Define Dictionary ADT and explain its purpose.
2. Differentiate between Dictionary ADT and Map ADT.
3. Why is a dictionary considered an abstract data type?
4. Explain the importance of key uniqueness in dictionaries.
5. What are the advantages of location-aware entries?

Programming Questions

1. Implement a dictionary using arrays in C++.
2. Modify dictionary implementation to support update operation.
3. Implement a dictionary using linked list.
4. Extend dictionary to handle duplicate values.
5. Implement a location-aware dictionary with delete functionality.

Conceptual Questions

1. Why is dictionary faster when implemented using hashing?
2. Compare dictionary ADT with binary search tree implementation.
3. What is the impact of using location tracking in dictionaries?
4. How does dictionary ADT relate to real-world databases?
5. Why is linear search used in simple dictionary implementations?

Key Takeaways

- Dictionary ADT stores **key-value pairs**
- Keys must be unique
- Can be implemented using arrays, lists, or hash tables
- Location-aware dictionaries track index positions
- Simple implementations use O(n) search
- Advanced implementations use hashing (O(1) average)
- Widely used in:
    - Databases

- Compilers (symbol tables)
- Search engines
- C++ STL map and unordered_map