We have combined Classroom and Theory tab and created a new Learn tab for easy access. You can access Classroom and Theory from the left panel.

Learn

Hashing - Inroduction

Hashing is a method of storing and retrieving data from a database efficiently.

Suppose that we want to design a system for storing employee records keyed using phone numbers. And we want the following queries to be performed efficiently:

1. Insert a phone number and the corresponding information.
2. Search a phone number and fetch the information.
3. Delete a phone number and the related information.

We can think of using the following data structures to maintain information about different phone numbers.

1. An array of phone numbers and records.
2. A linked list of phone numbers and records.
3. A balanced binary search tree with phone numbers as keys.
4. A direct access table.

For **arrays and linked lists**, we need to search in a linear fashion, which can be costly in practice. If we use arrays and keep the data sorted, then a phone number can be searched in O(Logn) time using Binary Search, but insert and delete operations become costly as we have to maintain sorted order.

With a **balanced binary search tree**, we get a moderate search, insert and delete time. All of these operations can be guaranteed to be in O(Logn) time.

Another solution that one can think of is to use a **direct access table** where we make a big array and use phone numbers as indexes in the array. An entry in the array is NIL if the phone number is not present, else the array entry stores pointer to records corresponding to the phone number. Time complexity wise this solution is the best of all, we can do all operations in O(1) time. For example, to insert a phone number, we create a record with details of the given phone number, use the phone number as an index and store the pointer to the record created in the table.
This solution has many practical limitations. The first problem with this solution is that the extra space required is huge. For example, if the phone number is of n digits, we need O(m * 10^n) space for the table where m is the size of a pointer to the record. Another problem is an integer in a programming language may not store n digits.

Due to the above limitations, the Direct Access Table cannot always be used. **Hashing** is the solution that can be used in almost all such situations and performs extremely well as compared to above data structures like Array, Linked List, Balanced BST in practice. With hashing, we get O(1) search time on average (under reasonable assumptions) and O(n) in the worst case.

> *Hashing is an improvement over Direct Access Table. The idea is to use a hash function that converts a given phone number or any other key to a smaller number and uses the small number as an index in a table called a hash table.*

Hash Function (http://en.wikipedia.org/wiki/Hash_function): A function that converts a given big phone number to a small practical integer

value. The mapped integer value is used as an index in the hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as an index in the hash table.
A good hash function should have following properties:

1. It should be efficiently computable.
2. It should uniformly distribute the keys (Each table position be equally likely for each key).

For example, for phone numbers, a bad hash function is to take the first three digits. A better function will consider the last three digits. Please note that this may not be the best hash function. There may be better ways.

Hash Table (http://en.wikipedia.org/wiki/Hash_table): An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry.

Collision Handling: Since a hash function gets us a small number for a big key, there is a possibility that two keys result in the same value. The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision and must be handled using some collision handling technique. Following are the ways to handle collisions:

- **Chaining:**The idea is to make each cell of the hash table point to a linked list of records that have the same hash function value. Chaining is simple, but it requires additional memory outside the table.
- **Open Addressing:** In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine the table slots until the desired element is found or it is clear that the element is not present in the table.

Open Addressing

Open Addressing: Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed).

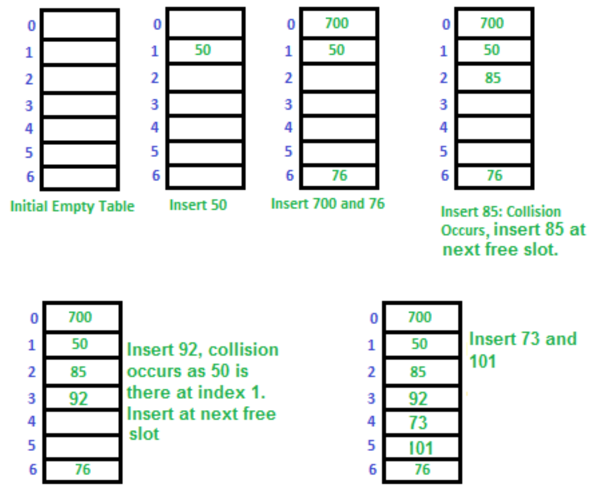
Important Operations:

- Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.
- Search(k): Keep probing until the slot's key doesn't become equal to k or an empty slot is reached.
- Delete(k): ***Delete operation is interesting***. If we simply delete a key, then the search may fail. So slots of the deleted keys are marked specially as "deleted".

Insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.

Open Addressing is done in the following ways:

1. ***Linear Probing:*** In linear probing, we linearly probe for the next slot. For example, the typical gap between the two probes is 1 as taken in the below example also.
   let **hash(x)** be the slot index computed using a hash function and **S** be the table size.
   ```
   If slot hash(x) % S is full, then we try (hash(x) + 1) % S
   If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S
   If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S
   ..................................................
   ..................................................
   ```
   Let us consider a simple hash function as “key mod 7” and a sequence of keys as 50, 700, 76, 85, 92, 73, 101.


Initial Empty Table
Insert 50
Insert 700 and 76
Insert 85: Collision Occurs, insert 85 at next free slot.
Insert 92, collision occurs as 50 is there at index 1. Insert at next free slot
Insert 73 and 101

(https://media.geeksforgeeks.org/wp-content/cdn-uploads/gq/2015/08/openAddressing1.png)

Clustering: The main problem with linear probing is clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element.

2. ***Quadratic Probing*** We look for i^2'th slot in i'th iteration.

```
let hash(x) be the slot index computed using hash function.
If slot hash(x) % S is full, then we try (hash(x) + 1*1) % S
If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S
If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S
..................................................
..................................................
```

3. **Double Hashing (https://www.cdn.geeksforgeeks.org/double-hashing/)** We use another hash function hash2(x) and look for i*hash2(x) slot in i'th rotation.

```
let hash(x) be the slot index computed using hash function.
If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S
If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S
If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S
..................................................
..................................................
```

See this (https://www.cse.cuhk.edu.hk/irwin.king/_media/teaching/csc2100b/tu6.pdf)for step by step diagrams.

Comparison of above three:

- Linear probing has the best cache performance but it suffers from clustering. One more advantage of Linear probing that it is easy to compute.
- Quadratic probing lies between the two in terms of cache performance and clustering.
- Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

S.No.	**Seperate Chaining**	**Open Addressing**
1.	Chaining is Simpler to implement.	Open Addressing requires more computation.
2.	In chaining, Hash table never fills up, we can always add more elements to chain.	In open addressing, table may become full.
3.	Chaining is Less sensitive to the hash function or load factors.	Open addressing requires extra care for to avoid clustering and load factor.
4.	Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.	Open addressing is used when the frequency and number of keys is known.
5.	Cache performance of chaining is not good as keys are stored using linked list.	Open addressing provides better cache performance as everything is stored in the same table.
6.	Wastage of Space (Some Parts of hash table in chaining are never used).	In Open addressing, a slot can be used even if an input doesn't map to it.
7.	Chaining uses extra space for links.	No links in Open addressing

Performance of Open Addressing: Like Chaining, the performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of the table (simple uniform hashing).

```
m = Number of slots in the hash table
n = Number of keys to be inserted in the hash table

Load factor α = n/m  ( < 1 )

Expected time to search/insert/delete < 1/(1 - α)

So Search, Insert and Delete take (1/(1 - α)) time
```

- Hashing in C++ using STL

Hashing in C++ can be implemented using different containers present in STL as per the requirement. Usually, STL offers the below-mentioned containers for implementing hashing:

- set
- unordered_set
- map
- unordered_map

Let us take a look at each of these containers in details:

set

Sets are a type of associative containers in which each element has to be unique, because the value of the element identifies it. The value of the element cannot be modified once it is added to the set, though it is possible to remove and add the modified value of that element.

Sets are used in the situation where it is needed to check if an element is present in a list or not. It can also be done with the help of arrays, but it would take up a lot of space. Sets can also be used to solve many problems related to sorting as the elements in the set are arranged in a sorted order.

Some basic functions associated with Set:

- **begin()** – Returns an iterator to the first element in the set.
- **end()** – Returns an iterator to the theoretical element that follows last element in the set.
- **size()** – Returns the number of elements in the set.
- **insert(val)** – Inserts a new element *val* in the Set.
- **find(val)** - Returns an iterator pointing to the element *val* in the set if it is present.
- **empty()** – Returns whether the set is empty.

Implementation:

```

// C++ program to illustrate hashing using
// set in CPP STL

#include <iostream>
#include <set>
#include <iterator>

using namespace std;

int main()
{
    // empty set container
    set <int> s;

    // List of elements
    int arr[] = {40, 20, 60, 30, 50, 50, 10};

    // Insert the elements of the List
    // to the set
    for(int i = 0; i < 7; i++)
        s.insert(arr[i]);

    // Print the content of the set
    // The elements of the set will be sorted
    // without any duplicates
    cout<<"The elements in the set are: \n";
    for(auto itr=s.begin(); itr!=s.end(); itr++)
    {
        cout<<*itr<<" ";
```

Run

Output:

```
The elements in the set are:
10 20 30 40 50 60

50 is present
```

LIVE BATCHES

unordered_set

The unordered_set container is implemented using a hash table where keys are hashed into indices of this hash table so it is not possible to maintain any order. All operation on unordered_set takes constant time O(1) on an average which can go up to linear time in the worst case which depends on the internally used hash function but practically they perform very well and generally provide constant time search operation.

The unordered-set can contain key of any type – predefined or user-defined data structure but when we define key of a user-defined type, we need to specify our comparison function according to which keys will be compared.

set vs unordered_set

- Set is an ordered sequence of unique keys whereas unordered_set is a set in which key can be stored in any order, so unordered.
- Set is implemented as a balanced tree structure that is why it is possible to maintain order between the elements (by specific tree traversal). The time complexity of set operations is O(Log n) while for unordered_set, it is O(1).

Note: Like set containers, the Unordered_set also allows only unique keys.

Implementation:

```

// C++ program to illustrate hashing using
// unordered_set in CPP STL

#include <iostream>
#include <unordered_set>
#include <iterator>

using namespace std;

int main()
{
    // empty set container
    unordered_set <int> s;

    // List of elements
    int arr[] = {40, 20, 60, 30, 50, 50, 10};

    // Insert the elements of the List
    // to the set
    for(int i = 0; i < 7; i++)
        s.insert(arr[i]);

    // Print the content of the unordered set
    // The elements of the set will not be sorted
    // without any duplicates
    cout<<"The elements in the unordered_set are: \n";
    for(auto itr=s.begin(); itr!=s.end(); itr++)
    {
        cout<<*itr<<" ";
```

Run

```
The elements in the unordered_set are:
10 50 30 60 40 20

50 is present
```

Map container

As a set, the Map container is also associative and stores elements in an ordered way but Maps store elements in a mapped fashion. Each element has a key value and a mapped value. No two mapped values can have the same key values.

Some basic functions associated with Map:

- **begin()** – Returns an iterator to the first element in the map.
- **end()** – Returns an iterator to the theoretical element that follows last element in the map.
- **size()** – Returns the number of elements in the map.
- **empty()** – Returns whether the map is empty.
- **insert({keyvalue, mapvalue})** – Adds a new element to the map.
- **erase(iterator position)** – Removes the element at the position pointed by the iterator
- **erase(const g)**– Removes the key value 'g' from the map.
- **clear()** – Removes all the elements from the map.

LIVE BATCHES

Implementation:

```

// C++ program to illustrate Map container

#include <iostream>
#include <iterator>
#include <map>

using namespace std;

int main()
{
    // empty map container
    map<int, int> mp;

    // Insert elements in random order
    // First element of the pair is the key
    // second element of the pair is the value
    mp.insert(pair<int, int>(1, 40));
    mp.insert(pair<int, int>(2, 30));
    mp.insert(pair<int, int>(3, 60));
    mp.insert(pair<int, int>(4, 20));
    mp.insert(pair<int, int>(5, 50));
    mp.insert(pair<int, int>(6, 50));
    mp.insert(pair<int, int>(7, 10));

    // printing map mp
    map<int, int>::iterator itr;
    cout << "The map mp is : \n";
    cout << "KEY\tELEMENT\n";
    for (itr = mp.begin(); itr != mp.end(); ++itr) {
```

Run

Output:

```
The map mp is :
KEY    ELEMENT
1    40
2    30
3    60
4    20
5    50
6    50
7    10
```

unordered_map Container

The unordered_map is an associated container that stores elements formed by a combination of key value and a mapped value. The key value is used to uniquely identify the element and mapped value is the content associated with the key. Both key and value can be of any type predefined or user-defined.
Internally unordered_map is implemented using Hash Table, the key provided to map are hashed into indices of a hash table that is why the performance of data structure depends on hash function a lot but on an average, the cost of search, insert and delete from hash table is O(1).

Implementation:

```

// C++ program to demonstrate functionality of unordered_map
// C++ program to demonstrate functionality of unordered_map
#include <iostream>
#include <unordered_map>
using namespace std;

int main()
{
    // Declaring umap to be of <string, int> type
    // key will be of string type and mapped value will
    // be of double type
    unordered_map<string, int> umap;

    // inserting values
    umap.insert({"GeeksforGeeks", 10});
    umap.insert({"Practice", 20});
    umap.insert({"Contribute", 30});

    // Traversing an unordered map
      cout << "The map umap is : \n";
    cout << "KEY\t\tELEMENT\n";
    for (auto itr = umap.begin(); itr!= umap.end(); itr++)
      cout << itr->first
            << '\t' << itr->second << '\n';

      return 0;
}
```

Run

Output:

```
The map umap is :
KEY        ELEMENT
Contribute    30
GeeksforGeeks    10
Practice    20
```

unordered_map vs unordered_set

In unordered_set, we have only key, no value, these are mainly used to see presence/absence in a set. For example, consider the problem of counting frequencies of individual words. We can't use unordered_set (or set) as we can't store counts.

unordered_map vs map

map (like set) is an ordered sequence of unique keys whereas in the unordered_map key can be stored in any order, so unordered.
A map is implemented as a balanced tree structure that is why it is possible to maintain order between the elements (by specific tree traversal). The time complexity of map operations is O(Log n) while for unordered_set, it is O(1) on average.

- Implementing Hashing in Java

Java provides many built-in classes and interfaces to implement hashing easily. That is, without creating any HashTable or hash function. Java mainly provides us with the following classes to implement Hashing:

1. **HashTable (https://www.geeksforgeeks.org/java-util-hashtable-class-java/) (A synchronized implementation of hashing)**: This class implements a hash table, which maps keys to values. Any non-null object can be used as a key or as a value.

```

// Java program to demonstrate working of HashTable
import java.util.*;

class GFG {
    public static void main(String args[])
    {

        // Create a HashTable to store
        // String values corresponding to integer keys
        Hashtable<Integer, String>
            hm = new Hashtable<Integer, String>();

        // Input the values
        hm.put(1, "Geeks");
        hm.put(12, "forGeeks");
        hm.put(15, "A computer");
        hm.put(3, "Portal");

        // Printing the Hashtable
        System.out.println(hm);
    }
}

```

Run

LIVE BATCHES

Output:

```
{15=A computer, 3=Portal, 12=forGeeks, 1=Geeks}
```

2. **HashMap (https://www.geeksforgeeks.org/hashmap-treemap-java/) (A non-synchronized faster implementation of hashing)**:
HashMap is also similar to HashTables in Java but it is faster in comparison as it is not synchronized. HashMap is used to store key-value pairs or to map a given value to a given key. The general application of HashMap is to count frequencies of elements present in an array or a list.

```

// Java program to create HashMap from an array
// by taking the elements as Keys and
// the frequencies as the Values

import java.util.*;

class GFG {

    // Function to create HashMap from array
    static void createHashMap(int arr[])
    {
        // Creates an empty HashMap
        HashMap<Integer, Integer> hmap = new HashMap<Integer, Integer>();

        // Traverse through the given array
        for (int i = 0; i < arr.length; i++) {

            // Get if the element is present
            Integer c = hmap.get(arr[i]);

            // If this is first occurrence of element
            // Insert the element
            if (hmap.get(arr[i]) == null) {
                hmap.put(arr[i], 1);
            }

            // If elements already exists in hash map
            // Increment the count of element by 1
            else {
```

Run

Output:

```
{34=1, 3=1, 5=2, 10=3}
```

3. **LinkedHashMap (https://www.geeksforgeeks.org/hashmap-treemap-java/) (Similar to HashMap, but keeps order of elements)**:

```

// Java program to demonstrate working of LinkedHashMap
import java.util.*;

public class BasicLinkedHashMap
{
    public static void main(String a[])
    {
        LinkedHashMap<String, String> lhm =
                       new LinkedHashMap<String, String>();
        lhm.put("one", "practice.geeksforgeeks.org");
        lhm.put("two", "code.geeksforgeeks.org");
        lhm.put("four", "quiz.geeksforgeeks.org");

        // It prints the elements in same order
        // as they were inserted
        System.out.println(lhm);

        System.out.println("Getting value for key 'one': "
                                       + lhm.get("one"));
        System.out.println("Size of the map: " + lhm.size());
        System.out.println("Is map empty? " + lhm.isEmpty());
        System.out.println("Contains key 'two'? "+
                                  lhm.containsKey("two"));
        System.out.println("Contains value 'practice.geeks"
        +"forgeeks.org'? "+ lhm.containsValue("practice"+
        ".geeksforgeeks.org"));
        System.out.println("delete element 'one': " +
                           lhm.remove("one"));
        System.out.println(lhm);
```

Run

Output:

```
{one=practice.geeksforgeeks.org, two=code.geeksforgeeks.org, four=quiz.geeksforgeeks.org}
Getting value for key 'one': practice.geeksforgeeks.org
Size of the map: 3
Is map empty? false
Contains key 'two'? true
Contains value 'practice.geeksforgeeks.org'? true
delete element 'one': practice.geeksforgeeks.org
{two=code.geeksforgeeks.org, four=quiz.geeksforgeeks.org}
```

4. **HashSet (https://www.geeksforgeeks.org/hashset-in-java/) (Similar to HashMap, but maintains only keys, not pair)**: The HashSet class implements the Set interface, backed by a hash table which is actually a HashMap instance. The class also offers constant time performance for the basic operations like add, remove, contains, and size assuming that the hash function disperses the elements properly among the buckets. HashSet is generally used to keep a check on whether an element is present in a list or not.

```

// Java program to demonstrate working of HashSet
import java.util.*;

class Test {
    public static void main(String[] args)
    {
        HashSet<String> h = new HashSet<String>();

        // Adding elements into HashSet usind add()
        h.add("India");
        h.add("Australia");
        h.add("South Africa");
        h.add("India"); // adding duplicate elements

        // Displaying the HashSet
        System.out.println(h);

        // Checking if India is present or not
        System.out.println("\nHashSet contains India or not:"
                           + h.contains("India"));

        // Removing items from HashSet using remove()
        h.remove("Australia");

        // Printing the HashSet
        System.out.println("\nList after removing Australia:" + h);

        // Iterating over hash set items
        System.out.println("\nIterating over list:");
```

Run

Output:

```
[South Africa, Australia, India]

HashSet contains India or not:true

List after removing Australia:[South Africa, India]

Iterating over list:
South Africa
India
```

5. LinkedHashSet (https://www.geeksforgeeks.org/linkedhashset-class-in-java-with-examples/) (Similar to LinkedHashMap, but maintains only keys, not pair):

```java

// Java program to demonstrate working of LinkedHashSet

import java.util.LinkedHashSet;
public class Demo
{
    public static void main(String[] args)
    {
        LinkedHashSet<String> linkedset =
                           new LinkedHashSet<String>();

        // Adding element to LinkedHashSet
        linkedset.add("A");
        linkedset.add("B");
        linkedset.add("C");
        linkedset.add("D");

        // This will not add new element as A already exists
        linkedset.add("A");
        linkedset.add("E");

        System.out.println("Size of LinkedHashSet = " +
                                    linkedset.size());
        System.out.println("Original LinkedHashSet:" + linkedset);
        System.out.println("Removing D from LinkedHashSet: " +
                            linkedset.remove("D"));
        System.out.println("Trying to Remove Z which is not "+
                            "present: " + linkedset.remove("Z"));
        System.out.println("Checking if A is present=" +
                            linkedset.contains("A"));
```

Run

Output:

```
Size of LinkedHashSet = 5
Original LinkedHashSet:[A, B, C, D, E]
Removing D from LinkedHashSet: true
Trying to Remove Z which is not present: false
Checking if A is present=true
Updated LinkedHashSet: [A, B, C, E]
```

6. TreeSet (https://www.geeksforgeeks.org/treeset-in-java-with-examples/) (Implements the SortedSet interface, Objects are stored in a sorted and ascending order):

```java

// Java program to demonstrate working of TreeSet

import java.util.*;

class TreeSetDemo {
    public static void main(String[] args)
    {
        TreeSet<String> ts1 = new TreeSet<String>();

        // Elements are added using add() method
        ts1.add("A");
        ts1.add("B");
        ts1.add("C");

        // Duplicates will not get insert
        ts1.add("C");

        // Elements get stored in default natural
        // Sorting Order(Ascending)
        System.out.println("TreeSet: " + ts1);

        // Checking if A is present or not
        System.out.println("\nTreeSet contains A or not:"
                           + ts1.contains("A"));

        // Removing items from TreeSet using remove()
        ts1.remove("A");

        // Printing the TreeSet
```

LIVE BATCHES

Run

Output:

```
TreeSet: [A, B, C]

TreeSet contains A or not:true

TreeSet after removing A:[B, C]

Iterating over TreeSet:
B
C
```

Report An Issue

If you are facing any issue on this page. Please let us know.

geeksforgeeks/)

LIVE BATCHES