Double Hashing - GeeksforGeeks

Source: https://www.geeksforgeeks.org/double-hashing/

Courses Tutorials Practice Jobs DSA Tutorial Interview Questions Quizzes Must Do Advanced DSA System Design Aptitude Puzzles Interview Corner DSA Python Technical Scripter 2026 Explore DSA Fundamentals Logic Building Problems Analysis of Algorithms Data Structures Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Tutorial Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Segment Tree Binary Indexed Tree or Fenwick Tree Square Root (Sqrt) Decomposition Algorithm Binary Lifting Geometry Interview Preparation Interview Corner GfG160 Practice Problem GeeksforGeeks Practice - Leading Online Coding Platform Problem of The Day - Develop the Habit of Coding DSA Course 90% Refund Double Hashing Last Updated : 29 Mar, 2024 Double hashing is a collision resolution technique used in hash tables. It works by using two hash functions to compute two different hash values for a given key. The first hash function is used to compute the initial hash value, and the second hash function is used to compute the step size for the probing sequence. Double hashing has the ability to have a low collision rate, as it uses two hash functions to compute the hash value and the step size. This means that the probability of a collision occurring is lower than in other collision resolution techniques such as linear probing or quadratic probing. However, double hashing has a few drawbacks. First, it requires the use of two hash functions, which can increase the computational complexity of the insertion and search operations. Second, it requires a good choice of hash functions to achieve good performance. If the hash functions are not well-designed, the collision rate may still be high. Advantages of Double hashing The advantage of Double hashing is that it is one of the best forms of probing, producing a uniform distribution of records throughout a hash table. This technique does not yield any clusters. It is one of the effective methods for resolving collisions. Double hashing can be done using : (hash1(key) + i * hash2(key)) % TABLE_SIZE Here hash1() and hash2() are hash functions and TABLE_SIZE is size of hash table. (We repeat by increasing i when collision occurs) Method 1: First hash function is typically hash1(key) = key % TABLE_SIZE A popular second hash function is hash2(key) = PRIME - (key % PRIME) where PRIME is a prime smaller than the TABLE_SIZE. A good second Hash function is: It must never evaluate to zero Just make sure that all cells can be probed Below is the implementation of the above approach: CPP /* ** Handling of collision via open addressing ** Method for Probing: Double Hashing */ #include <iostream> #include <vector> #include <bitset> using namespace std ; #define MAX_SIZE 10000001ll class doubleHash { int TABLE_SIZE , keysPresent , PRIME ; vector < int > hashTable ; bitset < MAX_SIZE > isPrime ; /* Function to set sieve of Eratosthenes. */ void __setSieve (){ isPrime [0] = isPrime [1] = 1 ; for (long long i = 2 ; i * i <= MAX_SIZE ; i ++) if (isPrime [i] == 0) for (long long j = i * i ; j <= MAX_SIZE ; j += i) isPrime [j] = 1 ; } int inline hash1 (int value){ return value % TABLE_SIZE ; } int inline hash2 (int value){ return PRIME - (value % PRIME); } bool inline isFull (){ return (TABLE_SIZE == keysPresent); } public : doubleHash (int n){ __setSieve (); TABLE_SIZE = n ; /* Find the largest prime number smaller than hash table's size. */ PRIME = TABLE_SIZE - 1 ; while (isPrime [PRIME] == 1) PRIME -- ; keysPresent = 0 ; /* Fill the hash table with -1 (empty entries). */ for (int i = 0 ; i < TABLE_SIZE ; i ++) hashTable . push_back (-1); } void __printPrime (long long n){ for (long long i = 0 ; i <= n ; i ++) if (isPrime [i] == 0) cout << i << ", " ; cout << endl ; } /* Function to insert value in hash table */ void insert (int value){ if (value == -1 || value == -2){ cout << ("ERROR : -1 and -2 can't be inserted in the table \n "); } if (isFull ()){ cout << ("ERROR : Hash Table Full \n "); return ; } int probe = hash1 (value), offset = hash2 (value); // in linear probing offset = 1; while (hashTable [probe] != -1){ if (-2 == hashTable [probe]) break ; // insert at deleted element's location probe = (probe + offset) % TABLE_SIZE ; } hashTable [probe] = value ; keysPresent += 1 ; } void erase (int value){ /* Return if element is not present */ if (! search (value)) return ; int probe = hash1 (value), offset = hash2 (value); while (hashTable [probe] != -1) if (hashTable [probe] == value){ hashTable [probe] = -2 ; // mark element as deleted (rather than unvisited(-1)). keysPresent -- ; return ; } else probe = (probe + offset) % TABLE_SIZE ; } bool search (int value){ int probe = hash1 (value), offset = hash2 (value), initialPos = probe ; bool firstItr = true ; while (1){ if (hashTable [probe] == -1) // Stop search if -1 is encountered. break ; else if (hashTable

[probe] == value) // Stop search after finding the element. return true ; else if (probe == initialPos && ! firstItr) // Stop search if one complete traversal of hash table is completed. return false ; else probe = ((probe + offset) % TABLE_SIZE); // if none of the above cases occur then update the index and check at it. firstItr = false ; } return false ; } /* Function to display the hash table. */ void print (){ for (int i = 0 ; i < TABLE_SIZE ; i ++) cout << hashTable [i] << ", " ; cout << " \n " ; } }; int main (){ doubleHash myHash (13); // creates an empty hash table of size 13 /* Inserts random element in the hash table */ int insertions [] = { 115 , 12 , 87 , 66 , 123 }, n1 = sizeof (insertions) / sizeof (insertions [0]); for (int i = 0 ; i < n1 ; i ++) myHash . insert (insertions [i]); cout << "Status of hash table after initial insertions : " ; myHash . print (); /* ** Searches for random element in the hash table, ** and prints them if found. */ int queries [] = { 1 , 12 , 2 , 3 , 69 , 88 , 115 }, n2 = sizeof (queries) / sizeof (queries [0]); cout << " \n " << "Search operation after insertion : \n " ; for (int i = 0 ; i < n2 ; i ++) if (myHash . search (queries [i])) cout << queries [i] << " present \n " ; /* Deletes random element from the hash table. */ int deletions [] = { 123 , 87 , 66 }, n3 = sizeof (deletions) / sizeof (deletions [0]); for (int i = 0 ; i < n3 ; i ++) myHash . erase (deletions [i]); cout << "Status of hash table after deleting elements : " ; myHash . print (); return 0 ; } Java import java.util.BitSet ; import java.util.Vector ; class DoubleHash { private int TABLE_SIZE , keysPresent , PRIME ; private Vector < Integer > hashTable ; private BitSet isPrime ; private static final long MAX_SIZE = 10000001L ; /* Function to set sieve of Eratosthenes. */ private void setSieve () { isPrime . set (0 , true); isPrime . set (1 , true); for (long i = 2 ; i * i <= MAX_SIZE ; i ++) if (! isPrime . get ((int) i)) for (long j = i * i ; j <= MAX_SIZE ; j += i) isPrime . set ((int) j); } private int hash1 (int value) { return value % TABLE_SIZE ; } private int hash2 (int value) { return PRIME - (value % PRIME); } private boolean isFull () { return (TABLE_SIZE == keysPresent); } public DoubleHash (int n) { isPrime = new BitSet ((int) MAX_SIZE); setSieve (); TABLE_SIZE = n ; /* Find the largest prime number smaller than hash table's size. */ PRIME = TABLE_SIZE - 1 ; while (isPrime . get (PRIME)) PRIME -- ; keysPresent = 0 ; /* Fill the hash table with -1 (empty entries). */ hashTable = new Vector <> (); for (int i = 0 ; i < TABLE_SIZE ; i ++) hashTable . add (- 1); } private void printPrime (long n) { for (long i = 0 ; i <= n ; i ++) if (! isPrime . get ((int) i)) System . out . print (i + ", "); System . out . println (); } /* Function to insert value in hash table */ public void insert (int value) { if (value == - 1 || value == - 2) { System . out . println ("ERROR : -1 and -2 can't be inserted in the table"); } if (isFull ()) { System . out . println ("ERROR : Hash Table Full"); return ; } int probe = hash1 (value), offset = hash2 (value); // in linear probing offset = 1; while (hashTable . get (probe) != - 1) { if (- 2 == hashTable . get (probe)) break ; // insert at deleted element's location probe = (probe + offset) % TABLE_SIZE ; } hashTable . set (probe , value); keysPresent += 1 ; } public void erase (int value) { /* Return if element is not present */ if (! search (value)) return ; int probe = hash1 (value), offset = hash2 (value); while (hashTable . get (probe) != - 1) if (hashTable . get (probe) == value) { hashTable . set (probe , - 2); // mark element as deleted (rather than unvisited(-1)). keysPresent -- ; return ; } else probe = (probe + offset) % TABLE_SIZE ; } public boolean search (int value) { int probe = hash1 (value), offset = hash2 (value), initialPos = probe ; boolean firstItr = true ; while (true) { if (hashTable . get (probe) == - 1) // Stop search if -1 is encountered. break ; else if (hashTable . get (probe) == value) // Stop search after finding the element. return true ; else if (probe == initialPos && ! firstItr) // Stop search if one complete traversal of hash table is // completed. return false ; else probe = ((probe + offset) % TABLE_SIZE); // if none of the above cases occur then update the index and // check at it. firstItr = false ; } return false ; } /* Function to display the hash table. */ public void print () { for (int i = 0 ; i < TABLE_SIZE ; i ++) System . out . print (hashTable . get (i) + ", "); System . out . println (); } } public class Main { public static void main (String [] args) { DoubleHash myHash = new DoubleHash (13); // creates an empty hash table of size 13 /* Inserts random element in the hash table */ int [] insertions = { 115 , 12 , 87 , 66 , 123 }; int n1 = insertions . length ; for (int i = 0 ; i < n1 ; i ++) myHash . insert (insertions [i]); System . out . print ("Status of hash table after initial insertions : "); myHash . print (); /* ** Searches for random element in the hash table, ** and prints them if found. */ int [] queries = { 1 , 12 , 2 , 3 , 69 , 88 , 115 }; int n2 = queries . length ; System . out . println ("\n" + "Search operation after insertion : "); for (int i = 0 ; i < n2 ; i ++) if (myHash . search (queries [i])) System . out . println (queries [i] + " present"); /* Deletes random element from the hash table. */ int [] deletions = { 123 , 87 , 66 }; int n3 = deletions . length ; for (int i = 0 ; i < n3 ; i ++) myHash . erase (deletions [i]); System . out . print ("Status of hash table after deleting elements : "); myHash . print (); } } Python3 from typing import List import math MAX_SIZE = 10000001 class DoubleHash : def __init__ (self , n : int): self . TABLE_SIZE = n self . PRIME = self . __get_largest_prime (n - 1) self . keysPresent = 0 self . hashTable = [- 1] * n def __get_largest_prime (self , limit : int) -> int : is_prime = [True] * (limit + 1) is_prime [0], is_prime [1] = False , False for i in range (2 , int (math . sqrt (limit)) + 1): if

```
is_prime [ i ]: for j in range ( i * i , limit + 1 , i ): is_prime [ j ] = False for i in range ( limit , - 1 , - 1 ): if
is_prime [ i ]: return i def __hash1 ( self , value : int ) -> int : return value % self . TABLE_SIZE def
__hash2 ( self , value : int ) -> int : return self . PRIME - ( value % self . PRIME ) def is_full ( self ) ->
bool : return self . TABLE_SIZE == self . keysPresent def insert ( self , value : int ) -> None : if value ==
- 1 or value == - 2 : print ( "ERROR : -1 and -2 can't be inserted in the table" ) return if self . is_full ():
print ( "ERROR : Hash Table Full" ) return probe , offset = self . __hash1 ( value ), self . __hash2 (
value ) while self . hashTable [ probe ] != - 1 : if - 2 == self . hashTable [ probe ]: break probe = ( probe
+ offset ) % self . TABLE_SIZE self . hashTable [ probe ] = value self . keysPresent += 1 def erase ( self
, value : int ) -> None : if not self . search ( value ): return probe , offset = self . __hash1 ( value ), self .
__hash2 ( value ) while self . hashTable [ probe ] != - 1 : if self . hashTable [ probe ] == value : self .
hashTable [ probe ] = - 2 self . keysPresent -= 1 return else : probe = ( probe + offset ) % self .
TABLE_SIZE def search ( self , value : int ) -> bool : probe , offset , initialPos , firstItr = self . __hash1 (
value ), self . __hash2 ( value ), self . __hash1 ( value ), True while True : if self . hashTable [ probe ] ==
- 1 : break elif self . hashTable [ probe ] == value : return True elif probe == initialPos and not firstItr :
return False else : probe = ( probe + offset ) % self . TABLE_SIZE firstItr = False return False def print (
self ) -> None : print ( * self . hashTable , sep = ', ' ) if __name__ == '__main__' : myHash =
DoubleHash ( 13 ) # Inserts random element in the hash table insertions = [ 115 , 12 , 87 , 66 , 123 ] for
insertion in insertions : myHash . insert ( insertion ) print ( "Status of hash table after initial insertions : "
, end = "" ) myHash . print () # Searches for random element in the hash table, and prints them if found.
queries = [ 1 , 12 , 2 , 3 , 69 , 88 , 115 ] n2 = len ( queries ) print ( " \n Search operation after insertion : "
) for i in range ( n2 ): if myHash . search ( queries [ i ]): print ( queries [ i ], "present" ) # Deletes random
element from the hash table. deletions = [ 123 , 87 , 66 ] n3 = len ( deletions ) for i in range ( n3 ):
myHash . erase ( deletions [ i ]) print ( "Status of hash table after deleting elements : " , end = '' )
myHash . print () C# using System ; using System.Collections.Generic ; using System.Linq ; class
doubleHash { int TABLE_SIZE , keysPresent , PRIME , MAX_SIZE = 10000001 ; List < int > hashTable
; bool [] isPrime ; /* Function to set sieve of Eratosthenes. */ void __setSieve () { isPrime [ 0 ] = isPrime [
1 ] = true ; for ( long i = 2 ; i * i <= MAX_SIZE ; i ++ ) { if ( isPrime [ i ] == false ) { for ( long j = i * i ; j <=
MAX_SIZE ; j += i ) { isPrime [ j ] = true ; } } } } int hash1 ( int value ) { return value % TABLE_SIZE ; } int
hash2 ( int value ) { return PRIME - ( value % PRIME ); } bool isFull () { return ( TABLE_SIZE ==
keysPresent ); } public doubleHash ( int n ) { isPrime = new bool [ MAX_SIZE + 1 ]; __setSieve ();
TABLE_SIZE = n ; /* Find the largest prime number smaller than hash * table's size. */ PRIME =
TABLE_SIZE - 1 ; while ( isPrime [ PRIME ] == true ) PRIME -- ; keysPresent = 0 ; hashTable = new
List < int > (); /* Fill the hash table with -1 (empty entries). */ for ( int i = 0 ; i < TABLE_SIZE ; i ++ )
hashTable . Add ( - 1 ); } public void __printPrime ( long n ) { for ( long i = 0 ; i <= n ; i ++ ) if ( isPrime [ i
] == false ) Console . Write ( i + ", " ); Console . WriteLine (); } /* Function to insert value in hash table */
public void insert ( int value ) { if ( value == - 1 || value == - 2 ) { Console . Write ( "ERROR : -1 and -2
can't be inserted in the table\n" ); } if ( isFull ()) { Console . Write ( "ERROR : Hash Table Full\n" ); return
; } int probe = hash1 ( value ), offset = hash2 ( value ); // in linear probing offset = 1; while ( hashTable [
probe ] != - 1 ) { if ( - 2 == hashTable [ probe ]) break ; // insert at deleted element's // location probe = (
probe + offset ) % TABLE_SIZE ; } hashTable [ probe ] = value ; keysPresent += 1 ; } public void erase (
int value ) { /* Return if element is not present */ if ( ! search ( value )) return ; int probe = hash1 ( value
), offset = hash2 ( value ); while ( hashTable [ probe ] != - 1 ) if ( hashTable [ probe ] == value ) {
hashTable [ probe ] = - 2 ; // mark element as deleted (rather // than unvisited(-1)). keysPresent -- ;
return ; } else probe = ( probe + offset ) % TABLE_SIZE ; } public bool search ( int value ) { int probe =
hash1 ( value ), offset = hash2 ( value ), initialPos = probe ; bool firstItr = true ; while ( true ) { if (
hashTable [ probe ] == - 1 ) // Stop search if -1 is encountered. break ; else if ( hashTable [ probe ] ==
value ) // Stop search after finding // the element. return true ; else if ( probe == initialPos && ! firstItr ) //
Stop search if one // complete traversal of // hash table is // completed. return false ; else probe = ((
probe + offset ) % TABLE_SIZE ); // if none of the // above cases occur // then update the // index and
check // at it. firstItr = false ; } return false ; } /* Function to display the hash table. */ public void print () {
for ( int i = 0 ; i < TABLE_SIZE ; i ++ ) Console . Write ( hashTable [ i ] + ", " ); Console . Write ( "\n" ); } }
public class Program { static void Main () { doubleHash myHash = new doubleHash ( 13 ); // creates an
empty hash table of size 13 /* Inserts random element in the hash table */ int [] insertions = { 115 , 12 ,
87 , 66 , 123 }; int n1 = insertions . Length ; for ( int i = 0 ; i < n1 ; i ++ ) myHash . insert ( insertions [ i ]);
Console . Write ( "Status of hash table after initial insertions : " ); myHash . print (); /* ** Searches for
random element in the hash table, ** and prints them if found. */ int [] queries = { 1 , 12 , 2 , 3 , 69 , 88 ,
115 }; int n2 = queries . Length ; Console . Write ( "\n" + "Search operation after insertion : \n" ); for ( int i
```

= 0 ; i < n2 ; i ++) if (myHash . search (queries [i])) Console . Write (queries [i] + " present\n"); /* Deletes random element from the hash table. */ int [] deletions = { 123 , 87 , 66 }; int n3 = deletions . Length ; for (int i = 0 ; i < n3 ; i ++) myHash . erase (deletions [i]); Console . Write ("Status of hash table after deleting elements : "); myHash . print (); } } JavaScript // JS code const MAX_SIZE = 10000001 ; // Set sieve of Eratosthenes let isPrime = new Array (MAX_SIZE). fill (0); isPrime [0] = isPrime [1] = 1 ; for (let i = 2 ; i * i <= MAX_SIZE ; i ++) { if (isPrime [i] === 0) { for (let j = i * i ; j <= MAX_SIZE ; j += i) { isPrime [j] = 1 ; } } } // Create DoubleHash Class class DoubleHash { constructor (n) { this . TABLE_SIZE = n ; this . PRIME = this . TABLE_SIZE - 1 ; while (isPrime [this . PRIME] === 1) { this . PRIME -- ; } this . keysPresent = 0 ; this . hashTable = new Array (this . TABLE_SIZE). fill (- 1); } isFull (){ return this . TABLE_SIZE == this . keysPresent ; } hash1 (value) { return value % this . TABLE_SIZE ; } hash2 (value) { return this . PRIME - (value % this . PRIME); } // Function to print prime numbers __printPrime (n) { for (let i = 0 ; i <= n ; i ++) { if (isPrime [i] === 0) { console . log (i + ", "); } } console . log ("\n"); } // Function to insert value in hash table insert (value) { if (value === - 1 || value === - 2) { console . log ("ERROR : -1 and -2 can't be inserted in the table\n"); } if (this . isFull ()) { console . log ("ERROR : Hash Table Full\n"); return ; } let probe = this . hash1 (value), offset = this . hash2 (value); // in linear probing offset = 1; while (this . hashTable [probe] !== - 1) { if (- 2 === this . hashTable [probe]) break ; // insert at deleted element's location probe = (probe + offset) % this . TABLE_SIZE ; } this . hashTable [probe] = value ; this . keysPresent += 1 ; } erase (value) { // Return if element is not present if (! this . search (value)) return ; let probe = this . hash1 (value), offset = this . hash2 (value); while (this . hashTable [probe] !== - 1) { if (this . hashTable [probe] === value) { this . hashTable [probe] = - 2 ; // mark element as deleted (rather than unvisited(-1)). this . keysPresent -- ; return ; } else { probe = (probe + offset) % this . TABLE_SIZE ; } } } search (value) { let probe = this . hash1 (value), offset = this . hash2 (value), initialPos = probe ; let firstItr = true ; while (1) { if (this . hashTable [probe] === - 1) break ; // Stop search if -1 is encountered. else if (this . hashTable [probe] === value) return true ; // Stop search after finding the element. else if (probe === initialPos && ! firstItr) return false ; // Stop search if one complete traversal of hash table is completed. else probe = (probe + offset) % this . TABLE_SIZE ; // if none of the above cases occur then update the index and check at it. firstItr = false ; } return false ; } // Function to display the hash table. print () { for (let i = 0 ; i < this . TABLE_SIZE ; i ++) console . log (this . hashTable [i] + ", "); console . log ("\n"); } } // Main function function main () { let myHash = new DoubleHash (13); // creates an empty hash table of size 13 // Inserts random element in the hash table let insertions = [115 , 12 , 87 , 66 , 123], n1 = insertions . length ; for (let i = 0 ; i < n1 ; i ++) myHash . insert (insertions [i]); console . log ("Status of hash table after initial insertions : "); myHash . print (); // Searches for random element in the hash table, and prints them if found. let queries = [1 , 12 , 2 , 3 , 69 , 88 , 115], n2 = queries . length ; console . log ("\n" + "Search operation after insertion : \n"); for (let i = 0 ; i < n2 ; i ++) if (myHash . search (queries [i])) console . log (queries [i] + " present\n"); // Deletes random element from the hash table. let deletions = [123 , 87 , 66], n3 = deletions . length ; for (let i = 0 ; i < n3 ; i ++) myHash . erase (deletions [i]); console . log ("Status of hash table after deleting elements : "); myHash . print (); return 0 ; } main (); // This code is contributed by ishankhandelwals. Output Status of hash table after initial insertions : -1, 66, -1, -1, -1, -1, 123, -1, -1, 87, -1, 115, 12,

Search operation after insertion : 12 present 115 present Status of hash table after deleting elements : -1, -2, -1, -1, -1, -1, -2, -1, -1, -2, -1, 115, 12, Time Complexity: Insertion: O(n) Search: O(n) Deletion: O(n) Auxiliary Space: O(size of the hash table). Comment Article Tags: Article Tags: Hash Technical Scripter DSA TCS-coding-questions Java-HijrahDate + 1 More