Dash
All
Articles
Videos
Problems
Quiz
Contest
Prev
Next

Implementation of Chaining

In hashing there is a hash function that maps keys to some values. But these hashing functions may lead to a collisio that is two or more keys are mapped to same value. **Chain hashing** avoids collision. The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

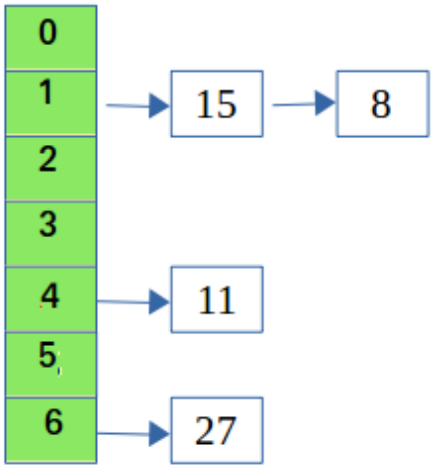
Let's create a hash function, such that our hash table has 'N' number of buckets.

To insert a node into the hash table, we need to find the hash index for the given key. And it could be calculated using the hash function.

Example: hashIndex = key % noOfBuckets
Insert: Move to the bucket corresponding to the above-calculated hash index and insert the new node at the end of the list.
Delete: To delete a node from hash table, calculate the hash index for the key, move to the bucket corresponding to the calculated hash index, and search the list in the current bucket to find and remove the node with the given key (if found).

Let's say hash table with 7 buckets (0, 1, 2, 3, 4, 5, 6)

Keys arrive in the Order (15, 11 , 27 , 8)



Please refer **Hashing | Set 2 (Separate Chaining)** for details.
We use a list in C++ which is internally implemented as linked list (Faster insertion and deletion).

Method - 1 :

This method has not concept of rehashing. It only has a fixed size array i.e. fixed numbers of buckets.

CPP

```
// CPP program to implement hashing with chaining
#include<bits/stdc++.h>
using namespace std;
```

Dash
All
Articles
Videos
Problems
Quiz
Contest

```
class Hash
{
    int BUCKET;    // No. of buckets

    // Pointer to an array containing buckets
    list<int> *table;
public:
    Hash(int V);  // Constructor

    // inserts a key into hash table
    void insertItem(int x);

    // deletes a key from hash table
    void deleteItem(int key);

    // hash function to map values to key
    int hashFunction(int x) {
        return (x % BUCKET);
    }

    void displayHash();
};

Hash::Hash(int b)
{
    this->BUCKET = b;
    table = new list<int>[BUCKET];
}

void Hash::insertItem(int key)
```

```
{
    int index = hashFunction(key);
    table[index].push_back(key);
}

void Hash::deleteItem(int key)
{
  // get the hash index of key
  int index = hashFunction(key);

  // find the key in (index)th list
  list <int> :: iterator i;
  for (i = table[index].begin();
           i != table[index].end(); i++) {
    if (*i == key)
      break;
  }

  // if key is found in hash table, remove it
  if (i != table[index].end())
    table[index].erase(i);
}

// function to display hash table
void Hash::displayHash() {
  for (int i = 0; i < BUCKET; i++) {
    cout << i;
    for (auto x : table[i])
      cout << " --> " << x;
    cout << endl;
```

```
    }
}

// Driver program
int main()
{
    // array that contains keys to be mapped
    int a[] = {15, 11, 27, 8, 12};
    int n = sizeof(a)/sizeof(a[0]);

    // insert the keys into the hash table
    Hash h(7);   // 7 is count of buckets in
                 // hash table
    for (int i = 0; i < n; i++)
        h.insertItem(a[i]);

    // delete 12 from hash table
    h.deleteItem(12);

    // display the Hash table
    h.displayHash();

    return 0;
}
```


Output

```
0
1 --> 15 --> 8
```

```
2
3
4 --> 11
5
6 --> 27
```


Time Complexity:

- **Search :** O(1+(n/m))
- **Delete :** O(1+(n/m))
  where n = Total elements in hash table
  m = Size of hash table
- Here n/m is the **Load Factor**.
- Load Factor (∝) must be as small as possible.
- If load factor increases,then possibility of collision increases.
- Load factor is trade of space and time .
- Assume , uniform distribution of keys ,
- Expected chain length : O(∝)
- Expected time to search : O(1 + ∝)
- Expected time to insert/ delete : O(1 + ∝)

Auxiliary Space: O(1), since no extra space has been taken.

Method - 2 :

Let's discuss another method where we have no boundation on number of buckets. Number of buckets will increase when value of load factor is greater than 0.5.

We will do rehashing when the value of load factor is greater than 0.5. In rehashing, we double the size of array and add all the values again to new array (doubled size array is new array) based on hash function. Hash function should also be change as it is depends on number of buckets. Therefore, hash function behaves differently from the previous one.

- Our Hash function is : **(ascii value of character * some prime number ^ x) % total number of buckets.** In this case prime number is 31.
- Load Factor = number of elements in Hash Map / total number of buckets
- Our key should be string in this case.
- We can make our own Hash Function but it should be depended on the size of array because if we do rehashing then it must reflect changes and number of collisions should reduce.

C++

```
#include <iostream>
#define ll long long int

using namespace std;

// Linked List
template <typename T>
class node
{
public:
    string key;
    T value;
    node *next;

    node(string key, T value) // constructor
    {
        this->key = key;
        this->value = value;
        this->next = NULL;
    }
```

```
    node(node &obj) // copy constructor
    {
        this->key = obj.key;
        this->value = obj.value;
        this->next = NULL;
    }

    ~node() // destructor
    {
        node *head = this;
        while (head != NULL)
        {
            node *currNode = head;
            head = head->next;
            delete currNode;
        }
    }
};

// hash table
template <typename T>
class unordered_map
{
public:
    int numOfElements, capacity;
    node<T> **arr; // want a array which stores pointers to node<T> i.e. head of a Linked List

    unordered_map() // constructor
    {
        this->capacity = 1;
```

```
    this->numOfElements = 0;
    this->arr = new node<T> *[this->capacity];
    this->arr[0] = NULL;
}

int hashFunction(string key) // hash function for hashing a string
{
    int bucketIndex;
    ll sum = 0, factor = 31;
    for (int i = 0; i < key.size(); i++)
    {
          // sum = sum + (ascii value of character * (prime number ^ x)) % total number of buckets
          // factor = factor * prime number i.e. prime number ^ x
        sum = ((sum % this->capacity) + ((int(key[i])) * factor) % this->capacity) % this->capacity;
        factor = ((factor % INT16_MAX) * (31 % INT16_MAX)) % INT16_MAX;
    }

    bucketIndex = sum;
    return bucketIndex;
}

float getLoadFactor()
{
      // number of elements in hash table / total numbers of buckets
    return (float)(this->numOfElements + 1) / (float)(this->capacity);
}

void rehashing()
{
    int oldCapacity = this->capacity;
```

```
    node<T> **temp = this->arr; // temp is hodling current array

    this->capacity = oldCapacity * 2; // doubling the size of curren
    this->arr = new node<T> *[this->capacity]; // points to new array of doubled size

    for (int i = 0; i < this->capacity; i++)
    {
        arr[i] = NULL;
    }
    for (int i = 0; i < oldCapacity; i++) // copying all the previous values in new array
    {
        node<T> *currBucketHead = temp[i];
        while (currBucketHead != NULL) // copying whole linked list
        {
            this->insert(currBucketHead->key, currBucketHead->value); // insert function have now update
            currBucketHead = currBucketHead->next;
        }
    }
    delete[] temp; // deleting old array from heap memory
    return;
}

void insert(string key, T value)
{
    while (this->getLoadFactor() > 0.5f) // when load factor > 0.5
    {
        this->rehashing();
    }

    int bucketIndex = this->hashFunction(key);
```

```
        if (this->arr[bucketIndex] == NULL) // when there is no linked list at bucket
        {
            node<T> *newNode = new node<T>(key, value);
            arr[bucketIndex] = newNode;
        }
        else // adding at the head of current linked list
        {
            node<T> *newNode = new node<T>(key, value);
            newNode->next = this->arr[bucketIndex];
            this->arr[bucketIndex] = newNode;
        }
        return;
    }

    int search(string key)
    {
        int bucketIndex = this->hashFunction(key); // getting bucket index
        node<T> *bucketHead = this->arr[bucketIndex];
        while (bucketHead != NULL) // searching in the linked list which is present at bucket for given key
        {
            if (bucketHead->key == key)
            {
                return bucketHead->value;
            }
            bucketHead = bucketHead->next; // moving to next node in linked list
        }
        cout << "Oops!! Data not found." << endl; // when key is not matched...
        return -1;
    }
};
```

```
{
    unordered_map<int> mp; // int is value....in our case key must be of string type
      mp.insert("Manish", 16);
      mp.insert("Vartika", 14);
      mp.insert("ITT", 5);
      mp.insert("elite_Programmer", 4);
      mp.insert("pluto14", 14);
      mp.insert("GeeksForGeeks", 11);

      cout << "Value of GeeksForGeeks : " << mp.search("GeeksForGeeks") << endl;
      cout << "Value of ITT : " << mp.search("ITT") << endl;
      cout << "Value of Manish : " << mp.search("Manish") << endl;
      cout << "Value of Vartika : " << mp.search("Vartika") << endl;
      cout << "Value of elite_Programmer : " << mp.search("elite_Programmer") << endl;
      cout << "Value of pluto14 : " << mp.search("pluto14") << endl;

        // prints Oops!! Data not found and return -1
      mp.search("GFG"); // case when there is no key present in Hash Map..

      return 0;
}
```

Output

```
Value of GeeksForGeeks : 11
Value of ITT : 5
Value of Manish : 16
```

```
Value of Vartika : 14
Value of elite_Programmer : 4
Value of pluto14 : 14
Oops!! Data not found.
```


Complexity analysis of Insert:

- **Time Complexity:** O(N), It takes O(N) time complexity because we are checking the load factor each time and whe it is greater than 0.5 we call rehashing function which takes O(N) time.
- **Space Complexity**: O(N), It takes O(N) space complexity because we are creating a new array of doubled size and copying all the elements to the new array.

Complexity analysis of Search:

- **Time Complexity:** O(N), It takes O(N) time complexity because we are searching in a linked list of size N.
- **Space Complexity:** O(1), It takes O(1) space complexity because we are not using any extra space for searching.