28/05/2023

Hashmaps and Tries

Hashmap is a datastructure which stores data in form of key-value pair.

Key	Value
Scorpio	9
Baleno	3

In the above example key is in the form of string and value is in the form of integer.

Note → Interviewer can say that tell some other solution other than map.

Types of map

1) Ordered map → In this time complexity is O(logn) for insertion, deletion and searching.

2) Unordered map → In this time complexity is O(1) for insertion, deletion and searching.

* A famous question in interview is that create a data structure in which insertion, deletion, updation and get random can be done in O(1).

STL implementation

To use map, we need to include a header file

```
#include <unordered_map>
```

(i) Creation

```
unordered_map <string, int> m;
```

string → Data type of key; int → Data type of value; m → name of map

(ii) Insertion

* pair <string, int> p = make_pair ("S", 9);
  m.insert (p);

* pair <string, int> p2 ("A", 10);
  m.insert (p2);

* m ["fortuner"] = 6;

(iii) Accessing

cout << m.at ("A"); → 10
↳ key

cout << m ["fortuner"]; → 6

(iv) Searching

* There is a count function to search for a key in a map.

cout << m.count ("A"); → 1
cout << m.count ("B"); → 0

1 means existing
0 means not existing

* find function → Returns iterator

```
if (m.find ("A") != m.end ()){
        cout << "Found";
}
else {
    cout << "Not found";
}
```

(v) size function

```
m.size();  → 3

m["A"] = 1;
m["B"] = 2;
m["C"] = 3;
m.size(); → 3
cout << m["D"] << endl; → 0
m.size(); → 4
```

m["D"] will create the entry of D having key as D and value as 0 and hence size of map becomes 4.

(vi) Iterating on map

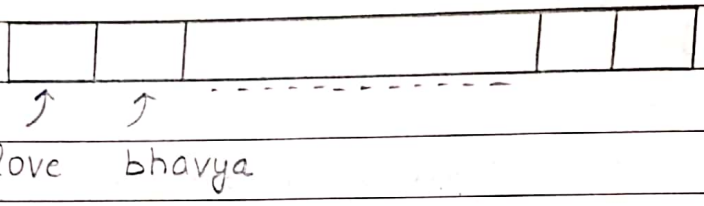
```
for (auto i : m) {
    cout << i.first << " ";
    cout << i.second << " ";
}
```

Note → As the name tells unordered, it is not necessary that the order is some as that of insertion.

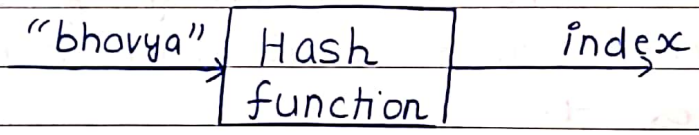
Implementation

1) We can implement with the help of linked list but time complexity will be O(n).
2) We can implement with the help of BST, but again in worst case for skewed tree, time complexity is O(n). For average case, its time complexity is O(logn) and ordered map is implemented via BST.

3) We can implement it with the help of array & we can perform the operations in O(1). Here we call array as bucket array.



We can do the above mapping with the help of hash function.



Hash function is made up of 2 components namely hash code & compression function.

* Hash code is responsible for conversion to some numeric value. It is not necessary that this numeric value is in range of array indexes.
* To bring the numeric value in the range, we have compression function.

bhavya ⟶ 2 + 8 + 1 + 22 + 25 + 1
= 59

59 is given to us via the hash code but now it is not in range. Hence the compression function is applied.

59 mod 26 = (7)
↳ in range now

Note - Hash function will give same result everytime for a particular key.

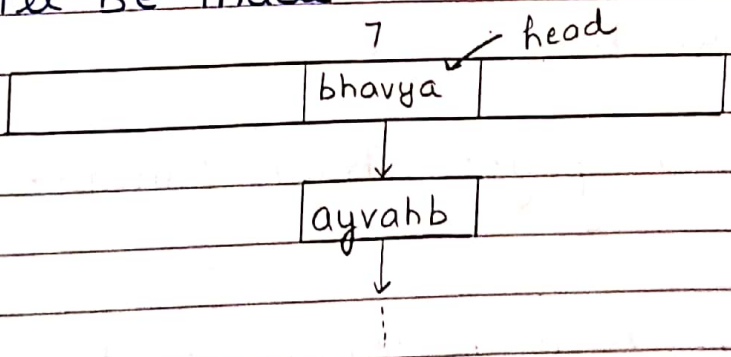
Collission
We need to have minimum collission in our hash function.

bhavya → 59 } Collission occurs
ayvabb → 59 }

Hence the hash function we used is a bad hash function.

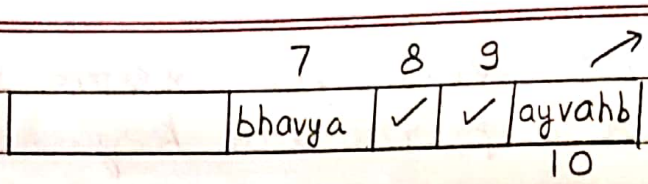
Handling the collission

(1) Open hashing
Suppose that at 7th index, bhavya is already present and now another string comes and has index = 7, so a linked list will be made.



But now time complexity is O(n) as linked list is made, but actually it is not O(n).

(2) Closed addressing
In this next free slot is searched and then at that free slot, value will be inserted



In this we are moving one step ahead & then checking for free slot. This is known as linear probing.

$$h(i) + f(i)$$

Here in linear probing, $f(i) = i$.

In quadratic probing, $f(i) = i^2$. That is initially keep $i = 1$, then move 1 step ahead & check for free. Then check for $i = 2$, that is $2^2 = 4$ & hence move 4 step ahead & check for free.

$$h(i) + f(i)$$
$$\hookrightarrow i^2$$

Good hash function
no. of elements = n
no. of free boxes = b

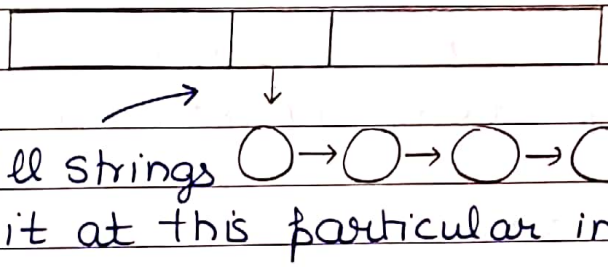
$\frac{n}{b} < 0.7$ } Good hash function

The above ratio is known as load factor.

Note → One question that comes to our mind is that when bhavya is passed to hash function, we are traversing it & hence here time complexity becomes $O(k)$ where k is size of string.

But here $n >> k$ and hence $O(k)$ is treated as constant time complexity.

Why time complexity in open hashing is not $O(n)$ inspite of using linked list?
We make our hash function so good, that the below case do not exist & hence $O(n)$ never arrives.



Q1 You are given a string. You have to tell the frequency of each character in the string.

Code
```
int main () {
string str = "bhavya";
unordered_map <char, int> freq;
for (int i=0 ; i< str.length () ; i++) {
        char ch = str [i];
        freq [ch] ++;
}
    return 0;
}
```

Q2 Check if a linked list is circular or not.
If a particular node is visited again, then a particular linked list is circular.

Code

```
bool checkCircular (Node * head){

    unordered_map <Node*, bool> vis;
    Node * temp = head;
    while (temp != NULL){
        // Not visited
        if (vis.find (temp) != vis.end())
                vis [temp] = true;
        else // Visited & hence return false
                return false;
        temp = temp → next; // Move temp ahead
    }
    return true;
}
```