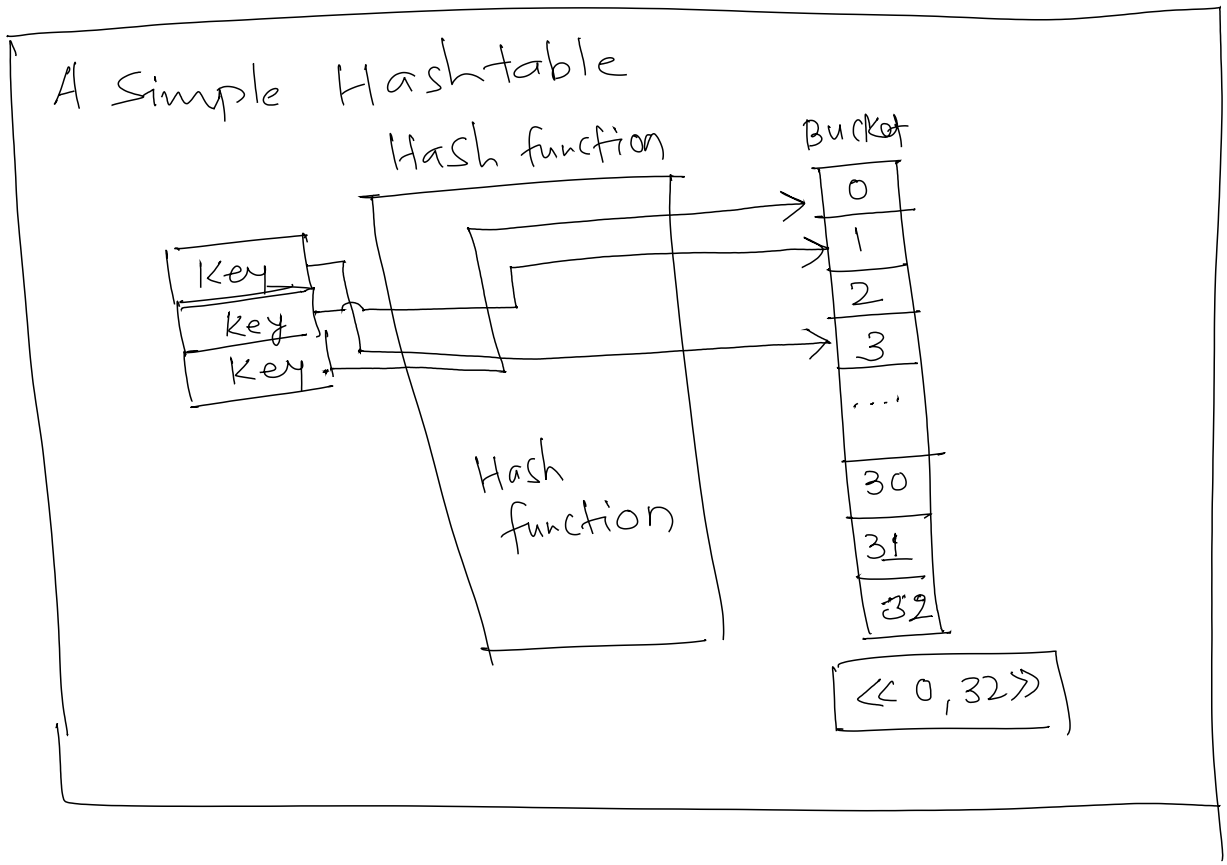
Hashing and Tries

15/07/2022
(1)

what is Hashing?

Hashing = Hash function (1) + Hash table (2)



Let's define Hash function?

"Mathematical function used to map some data (key) to a representative 'index'." This index is also referred to as a hash value

So, for ex:

A Simple Hash function

```
int hash(int key){
    int index = key % SIZE;
    return index;
}
```

what is a hash table?

"A hash table/map is a data structure that maps keys to values."

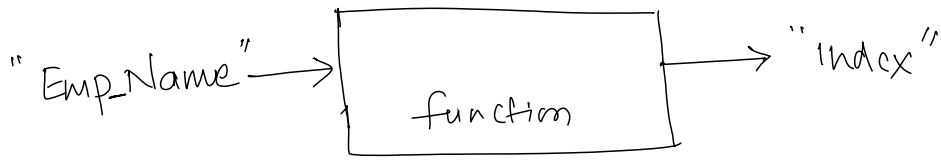
why do we use Hashing?

In general hashes are used to speedup searches.

As an example, Let's say you want to find the EmployeeID associated with a perticular Employee Name.

A naive approach is to search an entire database for the perticular employee. but this is not the optimal way to do it for large database.

so, to handle better of searching employee fooom database, hashing could be the alternative.



so, we have to use this function for generating the index of database and you will store the Employee info at this location, and someway you can search/retrieve also.

Hashing is the key concept in performant code for many search specific problems.

15/07/2022

Hashing could be a great bit if you can think of forming Convention & unconventional key to be used.

I mean, Conventional key Example : string, Integer etc.

Unconventional key example: matrix, list of integers (array, linked list etc.)

Please note Unconvention key finding is the bread/ butter of this approach.

Hash can sometime also be used to maintain states in certain questions. For example, to maintain visited states in a complicated BFS/DFS

How do we Implement Hashing ?

- Hashing would be achieved in traditional C++ by implementing a hash function that maps the key value to an integer.
- The hash table would then simply be an array of class/struct objects (the struct itself can be a list or other DS).

As an example

15/07/2022

④

Hash function for string: (DJB2)

①

```
unsigned long hash(unsigned char *str){
    unsigned long h = 5381;
    int c;
    while ( c = *str++)
    {
        h = ((h<<5) + h) + c;
    }
    return h % SIZE;
}
```

Hash fn for integers

②

```
Unsigned int hash(
    unsigned int x){
    x = ((x >> 16) ^ x) *
        0x45d9f3b;
    x = ((x >> 16) ^ x) *
        0x45d9f3b;
    x = (x >> 16) ^ x;
    return x % SIZE;
}
```

* Check the hash value here.

How do we implement hashing?

- Unordered maps are internally implemented as a hash table with a simple modulo hash function

- ordered maps are internally maintained as balanced binary tree (RB Tree) — therefore no hashing or collision occurs

For getting better decision, Please consider the below points

① How many bucket should I reserve.

② Should I use unordered_map or ordered_map

③ Is it possible to create a collision free hash function which I can use?

The Technical dept of Hashing

15/07/2022

(5)

Please not time complexity is not always O(1) as we understood from beginning.

- A major pitfall is the issue of collisions and collision handling.
- Users also need to be cognizant of the complexity of the hash function being used. (Similarly in ordered_map - the complexity of Compare function).
- Another pitfall is the blind reliance on a general purpose hash function — when a better / collision free hash function exists.

Let's understand these disadvantages in more detail.

~~What is Hash Collisions~~

How to work around Collision?

Chaining

- Chaining is the simple idea of storing a list of values at the index of a hash table — rather than a single value.
- The drawback is here you might need key also for verification of exact key match (Space Complexity increased)

15/07/2022
⑥

Open Addressing

- The idea is to find the next free space in the hash table to complete an insertion
- The free space can be searched for either in linear fashion or quadratic fashion.

> "Drawback": Here depending on the search pattern - our hash table would need to grow dynamically. (we will reach the load factor quickly and require a rehash)

Understanding collisions in unordered_maps

- unordered_maps work on the concept of "buckets". A "bucket" refers to the storage unit of one key-value pair.
- The unordered_map stores key-value pair by taking the modulo of input value by a prime number (107897 or 126271) and then stores it in hash table.
- When the input data is big and input values are multiple of this prime number a lot of collisions take place and may cause the complexity to O(n^2).
- Keeping in mind that it is prudent to change the hash function used by the unordered_map to a hash function with a better hash value distribution.

15/07/2022

(7)

The Bigger Enemy : Rehashing

When using a hash table we need to be aware of a concept known as the load_factor. This is specially important when using STL Unordered_maps.

What is load_factor?

Note

The load_factor is the percentage of used space in the container.

- Once the load_factor reaches a threshold known as max_load_factor a rehash/resize operation of the container is triggered.

- During a re-hash → All the elements in the container are rearranged according to their hash value into a new set of buckets. This may alter the order of iteration of elements within the container.

--- Complexity of rehash = O(n x Complexity of hash function) + "new memory allocation"

15/07/2022
(8)

How to prevent unnecessary rehash?

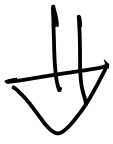
- A simple fix to the rehashing problem is to reserved your unordered_maps with the Unordered_map::reserve() operation

- Such reserve() operation is not available in ordered_map because ordered map is tree based and already optimized for memory uses. The concern of re-hashing is most prudent incase of unordered_map.

- bucket size of reserve method will be the maximum size of key-value pair in unordered_map

- In general use this reserve method for reserving sufficient (more than) to avoid un-necessary re-hashing.

Let's below compare bfn ordred_map vs unordred_map.



15/07/2022

09

Concept	std::map	std::unordered_map
Template Prototype	template <class key, class value, class Compare = less<key>>	template <class key, class value, class hash = hash<key>>
Internal Impl.	RBL Tree (self balancing binary tree)	Hash Table based on buckets (collision handled via open addressing)
Ordering	Sorted by key	unordered
Size	Effective for small and known key space	Effective for larger and Unknown key space
Average case (ins, del, search)	O(logn)	O(1)
Worst case	O(n)	O(n^k) where k is the length of key.
Memory Efficient	High memory Efficiency	Not memory efficient

Please note, sometime checking the performance b/n using unordered_map vs map is very useful. So please consider.