Hash Table

This chapter discusses operations on hash tables.

A hash table or hash map is essentially a *key, value* pair. The simplest way to think about a hash table is to imagine a row of houses in a neighborhood and a watchman for the neighborhood. You only know the names of the people in the neighborhood, but not their houses. You ask the watchman, where Mr. Zuckerberg lives and he points to house number 5. Next, you ask him where Mr. Gates lives and he points to house number 2. A hash table is similar in the sense that we'll give the data-structure a ***key***, and it will map our key to a slot where we can then place our ***value***.

The watchman maintains the mapping in his head, and the hash table will use a ***hash function*** to compute the mapping. Just like the watchman, the hash function will always point to the same slot when given the same key.

Array as a Hash Table

Let's say we are given a string of lowercase English-alphabet characters and are asked to find the frequency of each character in the given string. We can use an array of integers to act as our hash table.

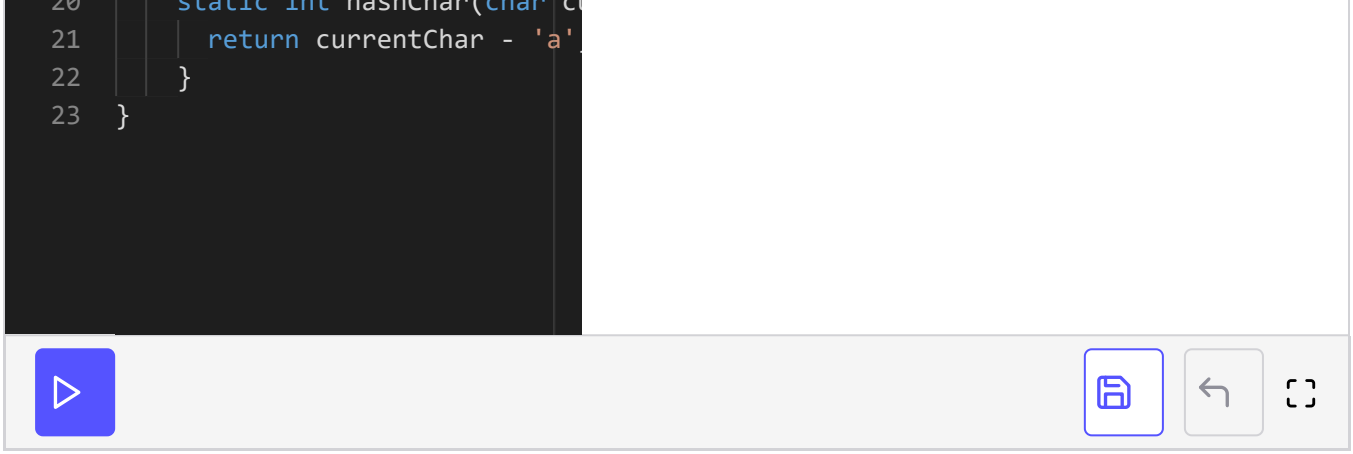
```
class HelloWorld {
    public static void main( St

      String givenStr = "aanal
      int[] hashTable = new int

      for(int i=0;i<givenStr.le
        char currentChar = give
        int key = hashChar(cur
        hashTable[key] = hashTa
      }

      for(int i=0;i<hashTable.l
        if(hashTable[i] !=0){
         System.out.println((ch
        }
      }
    }

    static int hashChar(char c
```



The hash function in our example is very simple. It merely subtracts 97='a' from the integer equivalent of the current character to normalize the index to one of the 26 slots.

Operations on Hash Table

- Let's talk about insertion first. Ask yourself, if the hash table consists of a million or a dozen entries, does it affect the number of steps to compute the hash of the key and place the value in the resulting slot? No, it doesn't. The hash function - **if assumed to take constant time to compute the hash** - isn't affected by the size of the table. Once the slot is determined, placing the value in the slot should also not be dependent on the size of the rest of the table and is, therefore, a constant time operation. Note, the assumption in bold is crucial for the hash table to maintain constant time operations.
- The hard part was the insertion operation, if that takes constant time then retrieval should be a breeze. We simply compute the hash of the given key, go to the right slot and retrieve the value. All-in-all, a constant number of steps allows for constant time.

When Rubber Meets the Road

If only the previous paragraphs were true in reality. Unfortunately, things get complicated when two different keys have their hashes pointing to the same slot. It's the same as the watchman saying Bill Gates and Jennifer Gates(daughter) live in house number 2. A **collision** is said to take place when two or more keys hash to the same slot. There are different ways to handle collisions, and each way will affect the complexity of different operations.

Collisions

If we design a naive hash table in which collisions are simply handled by a linked list, then insertion will still take constant time. Because, every time; a collision happens, we simply append to the head of the linked list maintained for the slot. However, imagine a really bad hash function, which hashes all keys to the same slot. In that scenario, you end up with a linked list masquerading as a hash table. Insertions are still O(1), but retrieval might mean going to the end of the linked list to retrieve a value. Therefore, the retrieval complexity becomes O(n) and defeats the purpose of using a hash table.

To mitigate the collision scenario, we can use a self-balancing binary search tree. If we are guaranteed by the hash function that a maximum of K keys can hash to the same slot, the retrieval in case of collision will take

$$log_2 K$$

time which is simply lgK. Note, we have sought an implicit guarantee from the hash function to work in constant time to come up with worst case retrieval complexity.

Hash Functions

There's a lot of science and mathematics behind creating good hash functions which are beyond the scope of this text. However, do remember that the hash function must take constant time to compute the hash of the given key. Also, collisions do happen and are expected. Your best bet for O(1), or constant time insert and retrieval operations is, to use hash tables provided by standard libraries in your language of choice.