# Hash Table

Hash Tables support insert, remove and search in O(1) time.

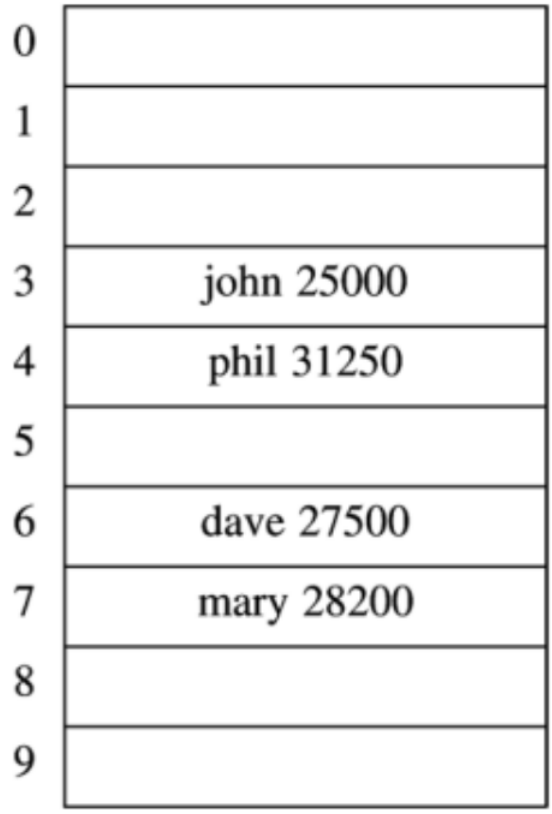
> **Important**
>
> **Idea Behind Hash Table:** O(1) is the worst case for insert, remove, search

## Issues that need to be solved

- Key may or may not be an integer
  - Need to use a function to map the key into an integer. This part of the functionality of a hash function. To hash the key into an index.
- Key space can be infinite
  - **Examples**: The key is a string of any length
  - Need to restrict the hash function to return a small value in order to index into the hash table
- Hash typically map a fairly large number
- **Use a hash function to map key**: integer or non integer to a relatively small integer as the index to the hash table
- To get good performance we must also use the hash function to evenly map keys into different indices of the has table

## Hashing

- Data items stored in an **array** of some fixed size
  - Hash table
- Search performed using some part of the data item
  - key
- Used for performing insertions, deletions, and finds in constant average time O(1)
- Operations requiring ordering information not supported efficiently




- The array has many unused entries.
- The array does not start from index 0.
- The index of an data item is computed from the data item.

## Applications of Hash Tables

- Comparing search efficiency of different data structures:
  - Vector, list: O(N)
  - Binary search tree: O(Log(N))
  - Hash table: O(1)
- C++ STL: `std::unordered_map`, `std::unordered_set`
- Compilers to keep track of declared variables
  - Symbol tables
- Game programs to keep track of positions visited
  - Transportation table
- On-line spelling checkers

## Hashing Functions

- Map keys to integers
- `Hash(key) = Integer`
- Evenly distributed index values
  - Even if the data is not evenly distributed
- **Assumptions**:
  - K: an unsigned 32 bit int
  - M: the number of buckets (the number of entries in a hash table)

- **Goal**:
  - If a bit is changed in K all bits are equally likely to change for Hash(K)
  - So that items evenly distributed in hash table

## Simple Function

- `Hash(K) = K % M`
- Where M is of any integer value
- Values of K may not be **evenly distributed**, however `Hash(k)` must be **evenly distributed**.
- **If** M = 10, K = 10, 20, 30, 40
- **Then** `K % M = 0, 0, 0, 0, 0 ...`

## Another Example

- `Hash(K) = K % P`
- Where P is Prime
- **Suppose** then P = 11, K = 10, 20, 30, 40
- `Then K % P = 10, 9, 8, 7`
- **A well designed hash table always has a prime number of entries**

## Hashing a Sequence of Keys

- K = $\{K_1, K_2, \ldots, K_n\}$
- `Hash("test") = 98157`
- Design Principles
  - Use the entire key
  - Use the ordering information

## Use the Entire Key

```C++
unsigned int Hash(const string& Key){
        unsigned int hash = 0;
        for(string::size_type k =0; j != K.size();++j)
        {
                hash = hash ^ Key[j]; // Xor
        }
        return hash;
}
// Problem: Hash("ab") == Hash("ba")
```

## Use the Ordering Information

```C++
unsigned int Hash(const string &Key){
        unsigned int hash = 0;
        for(size_type j =0; j != Key.size(); ++j)
        {
                hash = hash ^ Key[j];
                hash = hash * (j % 32);
        }
        return hash;
}
```

## Better Hash Function

```C++
unsigned int Hash(const string& S){
        size_type i;
        long unsigned int bigval = S[0];
        for(i = 1; i < S.size(); ++i){
                // low16 * magicNumber
                bigval = ((bigval & 65535) * 18000)
                + (bigcal >> 16) // high16
                + S[i];
        }
        bigval = ((bigval & 65535) * 18000) + (bigval >> 16);
        // bigval = low16 * magicNumber + high16

        // return low16
        return bigval & 65535;

}
```

Even if the function just returned 0 it would still be a legit hash function. Replacing the hash function in any hash table with this would still work but the O(1) complexity may not be maintained.