### Hash Table

## KEY/VALUE PAIR

| key | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| value | 10 | 9 | 15 | 14 | 1 | 12 | 13 |

- key can be defined to anything you want it to

For example, if you wanted the key to be a `"name"` (key) (it would search for) : `"Jack"`

Key is converted and mapped to be the index of the “array”, it then saves the value into the index

Hash table acts as array

index = (convert key to int) % array.size

### Time Complexities

insertion - O(1)
deletion - O(1)

### Keys

Let’s say we have the following keys,
“key0” : “Hello”
“name” : “Jack”

the following problem can occur:
“key0” → convert int → N % ArraySize = 4 → H(4) → 3;
”name” → convert int → M % ArraySize = 4;
“key1” → convert int → N % ArraySize = 4;

See, we have different keys, but the map to the same index. We can fix that with double hashing, except its not modern anymore, so we do chaining

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
|  |  |  |  | Hello | Jack | World |

### Chaining

Each index of the array points to its own linked list.
Always insert at the head so time complexity is O(1)
If you notice that an index’s linked list is too large (over the load factor), then you create a bigger array, so ArraySize increases and **rehash** keys

We can use a vector as well
HashTable Vector vector

```
insert(key, value)
int index = convertToIndex(key)
vector.insert(index, value)
```

Vector

```
insert(key, value)
index = convertToIndex(key)
        if(data[index] == null){
                data[index] = new SinglyLinkedList();
        }
        data[index].insertFront(key, value)
```