Hash Table 2

Recap of Hash Table 1 Notes

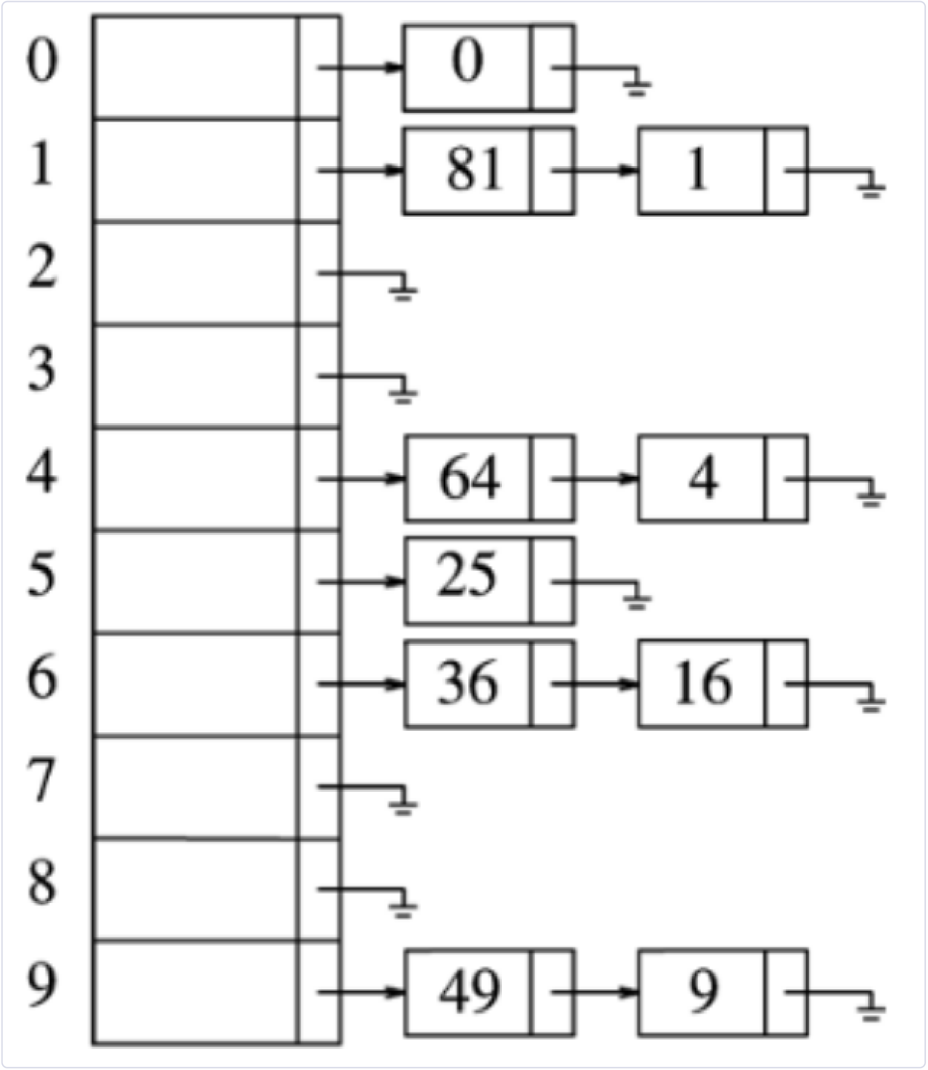
1. The basic idea of hash table is to approximate a giant array that is indexed by the key
2. A hash table is an array where the index of the data is computed (by the hash function) based on the key of the data

```
Index = hash(key) % table_size;
```

3. The situation when two keys are hashed into the same index is called a conflict or collision
4. A good hash function doesn't remove all conflicts. It statistically minimized the probability of collision across the key space

Separate Chaining

- Each table entry is a list - the hash table is physically an array of lists
- Multiple keys mapped to the same entry will be stored in the list



- **Assumptions**: `hash(k) = k % 10`
- Each `array[i]`, 0 ≤ i < size, is a list

Search with Separate Chaining

- To search for a key X we must do the following
  - `index = hash(X)`
  - Check list array `array[index]` to see if X is in it.
- Check the above graphic, using the assumptions for a value.

Inserting with Separate Chaining

- To insert key X we must do the following
  - `index = hash(X)`
  - Insert X into list `array[index]`
- Check the above graphic using the assumptions to insert a value

Delete with Separate Chaining

- To delete key X we must do the following
  - `index = hash(X)`
  - Insert X into list `array[index]`
- Check the above graphic using the assumptions to remove a value

Separate Chaining

- With separate chaining, the hash table is an array of containers
- Insertion, removal and deletion can be so quick because it only needs to do one calculation to find the location rather than comparing values
- The number of lists in the hash table needs to be roughly the same as the number of data items in the hash table
- The load factor (λ) of a hash table with separate chaining is the ratio of the number of elements in the table to the table size
- With separate chaining, the average list size is equal to λ!
- Typically, we want λ ≈ 1
- λ decides when to perform rehash (expanding the table)

Implementation

```C++
template <typename HashedObj>
class HashTable
{
public:
        explicit HashTable(int size=101);
        bool contains(HashedObj& x) const;

        void makeEmpty();
        bool insert(const HashedObj& x);
        bool insert(HashedObj&& x);
        bool remove(const HashedObj& x);

private:
        vector<list<HashedObj>> theLists;
        int currSize;

        void rehash();
        size_t myhash(const HashObj& x) const;
}
```

Hashed Object

```cpp
template <typename key>
class Hash {
public:
        size_t operator()(const key& k) const;

};
template <>
class Hash<string>{ // Implementation
public:
        size_t operator()(const string& key){
                // ...
        }
};
```

Class Example

```cpp
class Employee{
public:
        const string& getName() const{
                return name;
        }
        bool operator==(const Employee& rhs) const
        {
                return getName() == rhs.getName();
        }
        bool operator!=(const Employee& rhs)const {
                return !(*this == rhs);
        }
private:
        string name;
        double salary;
        int seniority;
        // Aditional private members
};
template<>
class hash<Employee>{
public:
        size_t operator()(const Employee& item){
                static hash<string> hf;
                return hf(item.getName());
        }
};
```

Separate Chaining

```cpp
// Separate Chaining
size_t myhash(const HashedObj& x){
        static hash<HashedObj> hf;
        return hf(x) % theList.size();
}

// Separate Chaining Cont'd
// More Function Definitions
void makeEmpty(){
        for(auto& theList: theList){
                theList.clear();
        }

}

bool contains(const HashedObj& x) const{
        auto & whichList = theList[myhash(x)];
        return find(begin(whichList), end(whichList) != end(whichList));
}

bool remove(const HashObj& x){
        auto& whichList = theList[myhash(x)];
        auto itr = find(begin(whichList), end(whichList), x);

        if(itr == end(whichList)){
                return false;
        }
        whichList.erase(itr);
        --currentSize;
        return true;
}

bool insert(const HashedObj& x){
        auto& whichList = theList[myhash(x)];
        if(find(begin(whichList), end(whichList), x) != end(whichList)){
                return false;
        }
        whichList.push_back(x);

        // rehash...
        if(++currentsize > theList.size()){
                rehash();
        }
        return true;
}
```

Hash Tables without Chaining

- Try to avoid buckets with separate list - no list, just an array of elements
- Still need to result conflicts - use Probing Hash Tables
  - If collision occurs, try another cell in the hash table.
  - More formally,, try cells $h_0(x), h_1(x), h_2(x), h_3(x), \ldots$ in succession until a free cell is found.
  - $h_i(x) = hash(x) + f(i)$
  - AND $f(0) = 0$

Linear Probing

- `f(i) = i`
  - Try `hash(x), hash(x) + 1, hash(x) + 2, ...`

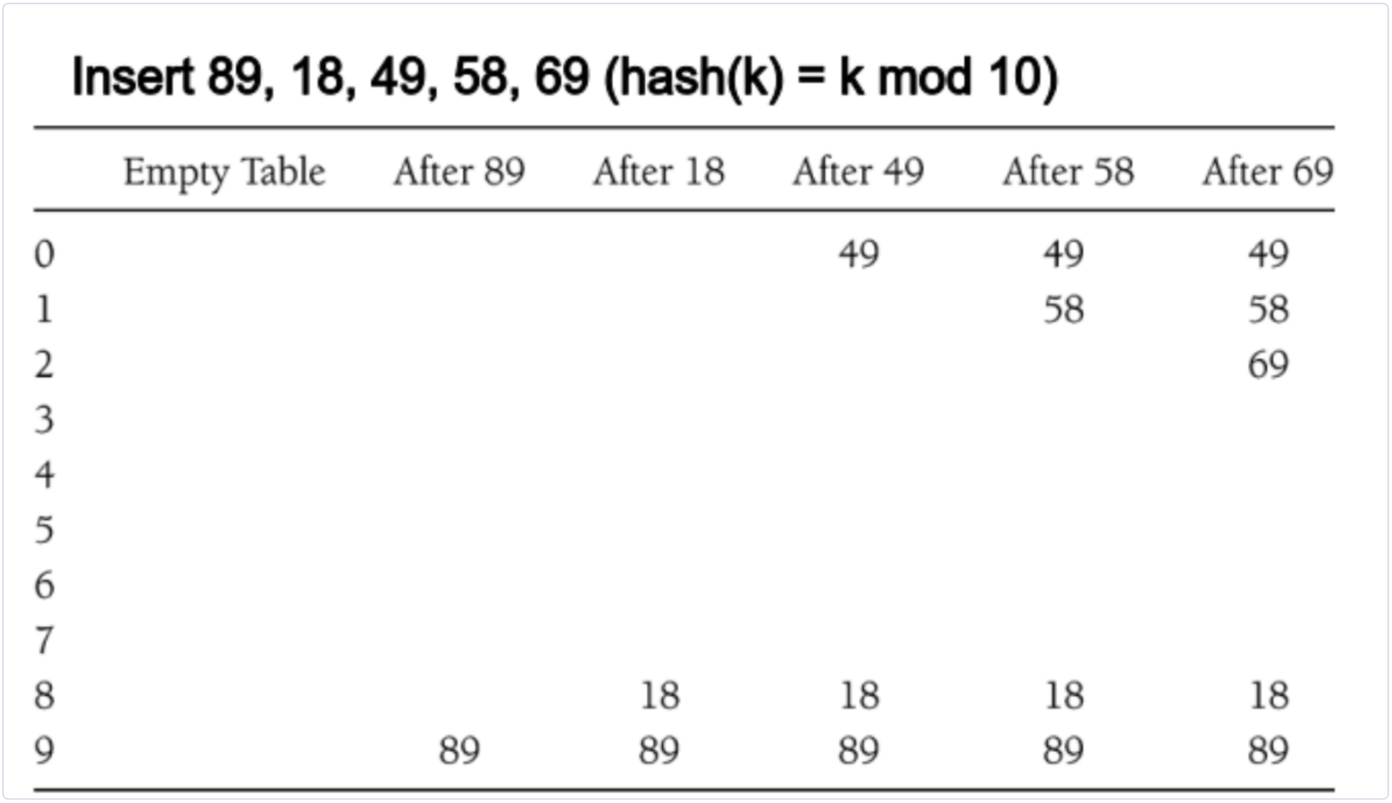
Insert (assume no duplicate keys)

1. `Index = hash(key) % table_size;`
2. If `table[index]` is empty, put informations (key and others) in entry `table[index]`

3. If `table[index]` is not empty then, `index++; index = index % table_size; goto 2`

Search (key)

1. `Index = hash(key) % table_size;`
2. If (`table[index]` is empty) return -1 (not found)
3. `Else if (table[index].key == key) return index;`
4. `index++; index = index % table_size; goto 2;`



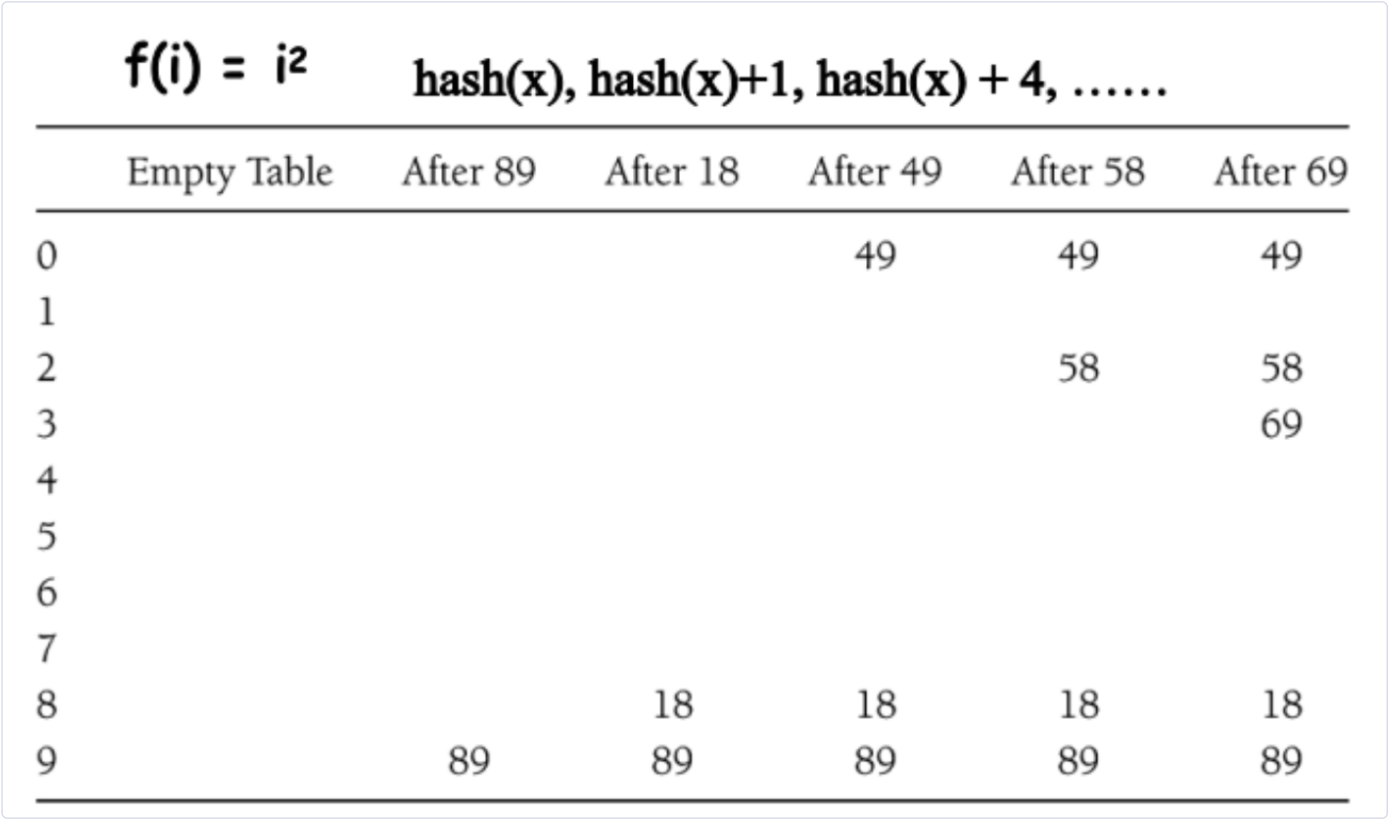
Insert 89, 18, 49, 58, 69 (hash(k) = k mod 10)

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Delete

- Can be tricky, must maintain the consistency of the hash table, consider the number 89 in the table above.
- What is the simplest deletion strategy you can think of??

Quadratic Probing



f(i) = i^2 hash(x), hash(x)+1, hash(x) + 4, ……

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

```C++
// Nested within Hash Table class
enum EntryType{
        ACTIVE,
        EMPTY,
        DELETED
};

struct HashEntry{
        HashedObj element;
        EntryType info;
        HashedEntry(const HashedObj& e = HashedObj(), EntryType ≠ EMPTY)
                : element{e}, info{|}{}
        HashEntry(HashedObj&& e, EntryType ≠ EMPTY)
                : element{std::move(e)}, info{|}{}
};
```

Double Hashing