Maps and Hash Tables

L.EIC

Algorithms and Data Structures / Algoritmos e Estruturas de Dados

AED

2023/2024

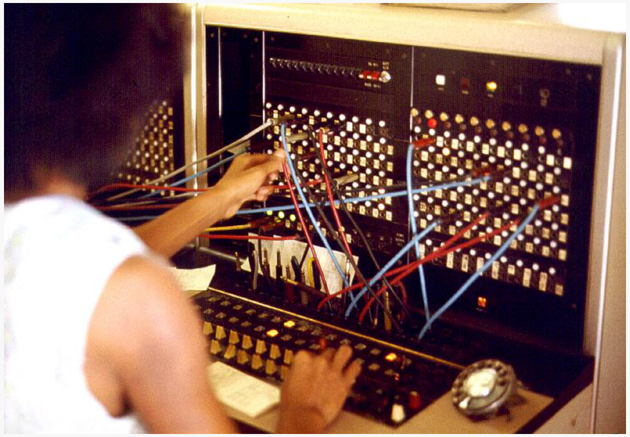
A.P. Rocha, P. Diniz,

A. Costa, B. Leite, F. Ramos, J. Pires, J. Oliveira, P. H. Diniz, V. Silva

A Bit of History: Telephone Switching

Placing Phone Call circa 1900

1. You Dial the Operator
2. Tell him/her the Person you would like to talk to
3. Operator Connects you to that Person (hopefully)



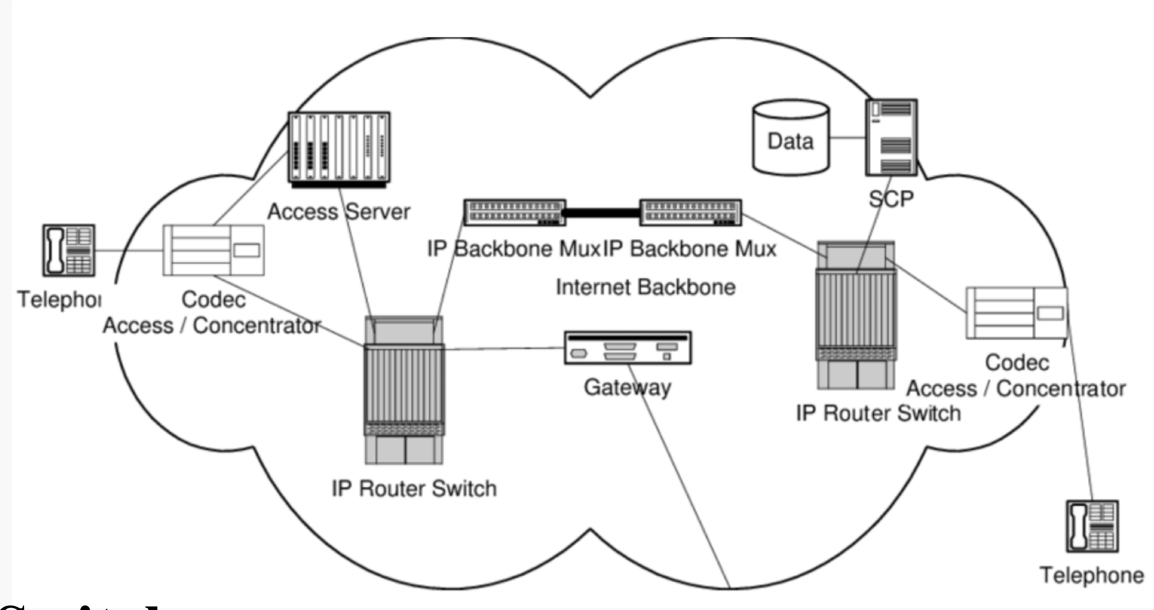
Operator

1. Has a List of Persons' Numbers
2. Maps each Person.s Phone Number to a Board Output
3. Connects Wires
4. Disconnect Wires at the end of your conversation (a light indicates when you are still talking)

Modern Version: Internet Packet Switching

Packet Switching

1. Given a Packet with a Destination Address
2. Need to Find to which Router/Channel to Send it To



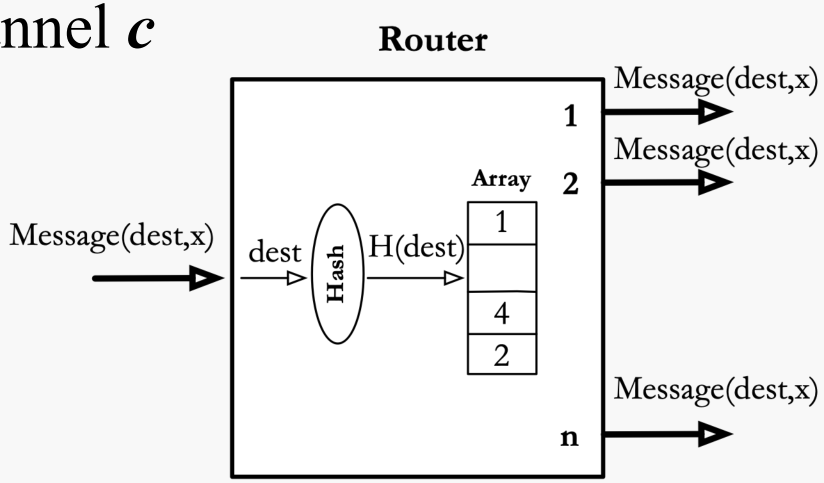
In Practice, at Each Switch:

1. Packet with Destination ***d***
2. Needs to Be Routed to Output Channel ***c***

Routing Implementation: Mapping Table

Basic Problem

1. Packet with Destination ***dest***, *Contents* ***x***
2. Routed to Router Output Channel ***c***



Implementation

- Keeps a Map of pairs (dest,c)
- Given a Packet
  1. Inspects the Destination (dest)
  2. Looks up the pair (dest,c)
  3. Sends the Packet to the next Router via Channel c

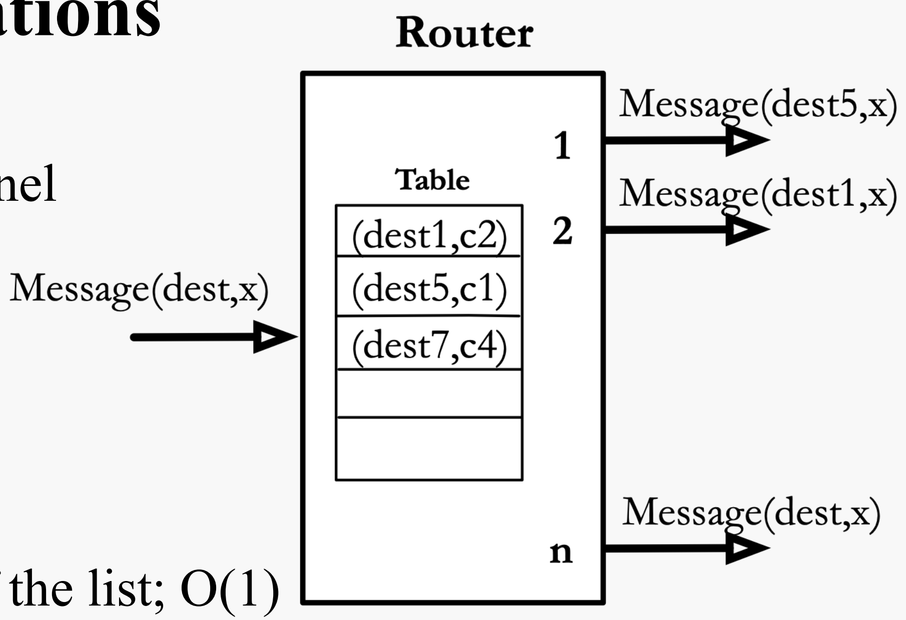
Key Issue

- Needs to be *Blazingly* Fast…

Routing Implementation: Mapping Table

Mapping Table Basic Operations

1. **Insert** a Mapping (dest, channel)
2. **Lookup** a Mapping (dest) → channel
3. **Remove** a Mapping (dest)



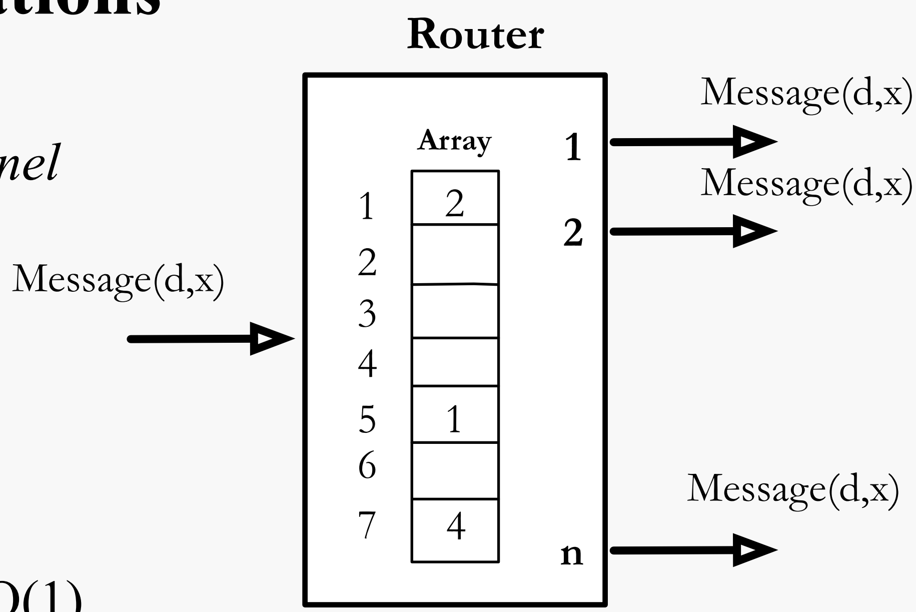
Implementations

- Linked List:
  - Insert – Append to the end of the list; O(1)
  - Lookup – Search the list; O(n)
  - Remove – Search and swap with last element of list; O(n)
- Binary Search Tree:
  - Insert – Traverse tree; insert and balance it; O(log n)
  - Lookup - If balanced; O(log n)
  - Remove – Look up and balance; O(log n)

Routing Implementation: Mapping Table

Mapping Table Basic Operations

1. **Insert** a Mapping (*dest*, *channel*)
2. **Lookup** a Mapping (*dest*) → *channel*
3. **Remove** a Mapping (*dest*)



Simple Table

- Use Array with *dest* as Index
  - Insert – Set table(*dest*) ← c; O(1)
  - Lookup – Index table(*dest*); O(1)
  - Remove – Reset table(*dest*) ← ∅; O(1)
- All Operations are O(1)

So, what is the big Deal?

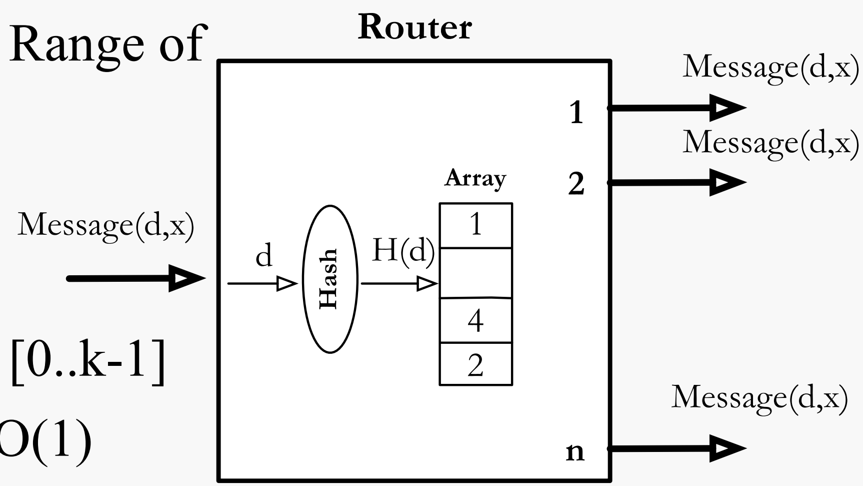
Routing Implementation: Hash Table

Observations:

1. Set of observed values in practice is small
2. Map Large Range of Keys to a Small Range of table indices

Hash Table:

- Use Array with size **k**
- Compute Hash Function: H(*dest*) → [0..k-1]
  - Insert – set table(H(*dest*)) ← c; O(1)
  - Lookup – index table(H(*dest*)); O(1)
  - Remove – reset table(H(*dest*)) ← ∅; O(1)
- All Operations are O(1)



Key Issues:

- What if H($dest_1$) = H($dest_2$)?
- Hash Function H must be "good"

Hash Table Implementation and Performance

Hash Table

- The *hash function* should:
  - Be easy to calculate, in the sense of computational cost
  - Evenly distribute objects across the table indices

- Multiple objects can be mapped to the same position: ***collision***

- The behavior of hash tables is characterized by:
  - Hash Function H
  - Collision Resolution Technique

- Good Hash tables ensure <u>constant average time</u> for insertion, removal and searching

Hash Table

Hash Function

Hash function takes into account the size of the table to ensure results are within the intended range

```
int hashString (const char* key, int tableSize) {
   int hashVal = 0;
   for (int i = 0; i < key.length ; i++)
      hashVal = 37*hashVal + key[i];
   hashVal %= tableSize;
   if (hashVal < 0)
      hashVal += tableSize;
   return hashVal;
}
```

```
int hashInt (int key, int tableSize) {
   if (key < 0) key = -key;
   return key % tableSize;
}
```

The quality of the hash function also depends on the size of the table: *prime* sizes are good candidate values.

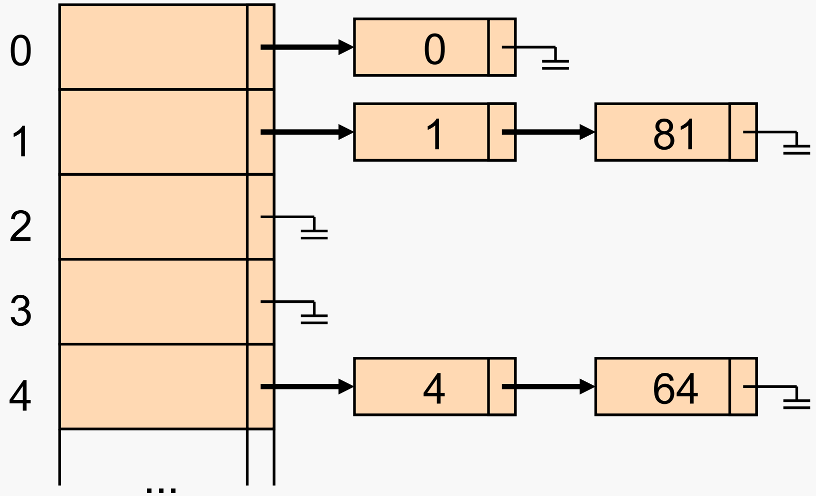
Collision & Resolution Strategies

- **Collision**
  - When Different Keys have the same Hash Value

- **Resolution Strategies** (using a Single Table)
  - Separate Chaining
  - Open Addressing
    - Linear Probing
    - Quadratic Probing
    - Double Hashing
    - Cuckoo Hashing
    - …

Collision Resolution: Separate Chaining

- **Strategy:**
  - Collisions resolved by listing all keys that map to the same hash value
  - Use Linked Lists.
  - Hash Table is an array of Linked Lists.
  - Elements in List Still need to have Data to Disambiguate Key or the original Key itself.



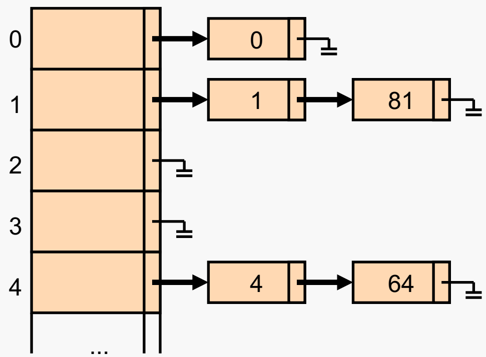
$Hash_i(x) = x \ \% \ 10$

Note: *In this example, table size =10 for simplicity (size should be a prime number)*

Collision Resolution: Separate Chaining

- **Performance:**
  - Measured by the number of probing performed.
    - Depends on the Load Factor λ (can have λ > 1)

$$\lambda = number\ of\ elements\ in\ the\ table / table\ size$$



  - Average Length of each List is λ
  - Average Search Time (number of probing)
    - Unsuccessful Search: λ
    - Successful Search: 1 + λ/2

Collision Resolution: Open Addressing

- **Strategy:**
  - Search for Alternative Positions in the Hash Table
  - How? probing the positions $Hash_1(x), Hash_2(x), \ldots$ until a free position is found.

$$\boxed{Hash_i(x) = \left(x + f(i)\right) \% \; tableSize}$$

  - If position not Found? **Insertion Fails…** Table is Full
  - Remove Operation: just Find it and Remove Item from Table… (*later*)

  - **Implementation Variants:**
    - **Linear Probing**: $f(i) = i$
      - Ensures full utilization of the table
    - **Quadratic Probing**: $f(i) = i^2$
      - It may be impossible to insert an element into a table, even with space
      - Avoids the phenomenon of primary aggregation

Collision Resolution: Open Addressing

- **Linear Probing Insertion Example**

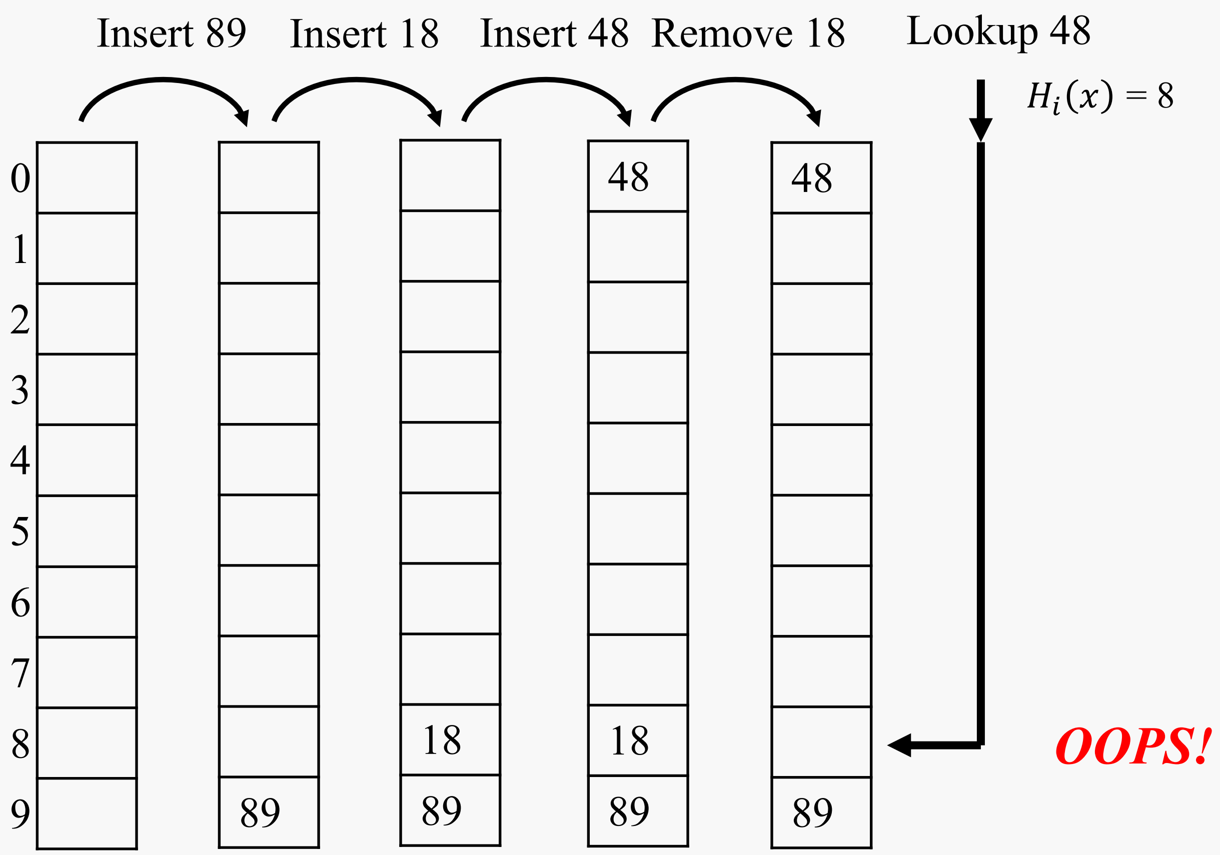
		Insert 89	Insert 18	Insert 49	Insert 58	Insert 69	Insert 21
0				49	49	49	49
1					58	58	58
2						69	69
3							21
4							
5							
6							
7							
8			18	18	18	18	18
9		89	89	89	89	89	89

$$H_i(x) = (x + i) \% 10$$

Note: *in the example, table size =10*
just for simplification (as this should be a prime number)

Collision Resolution: Open Addressing

- **Open Addressing Removal**



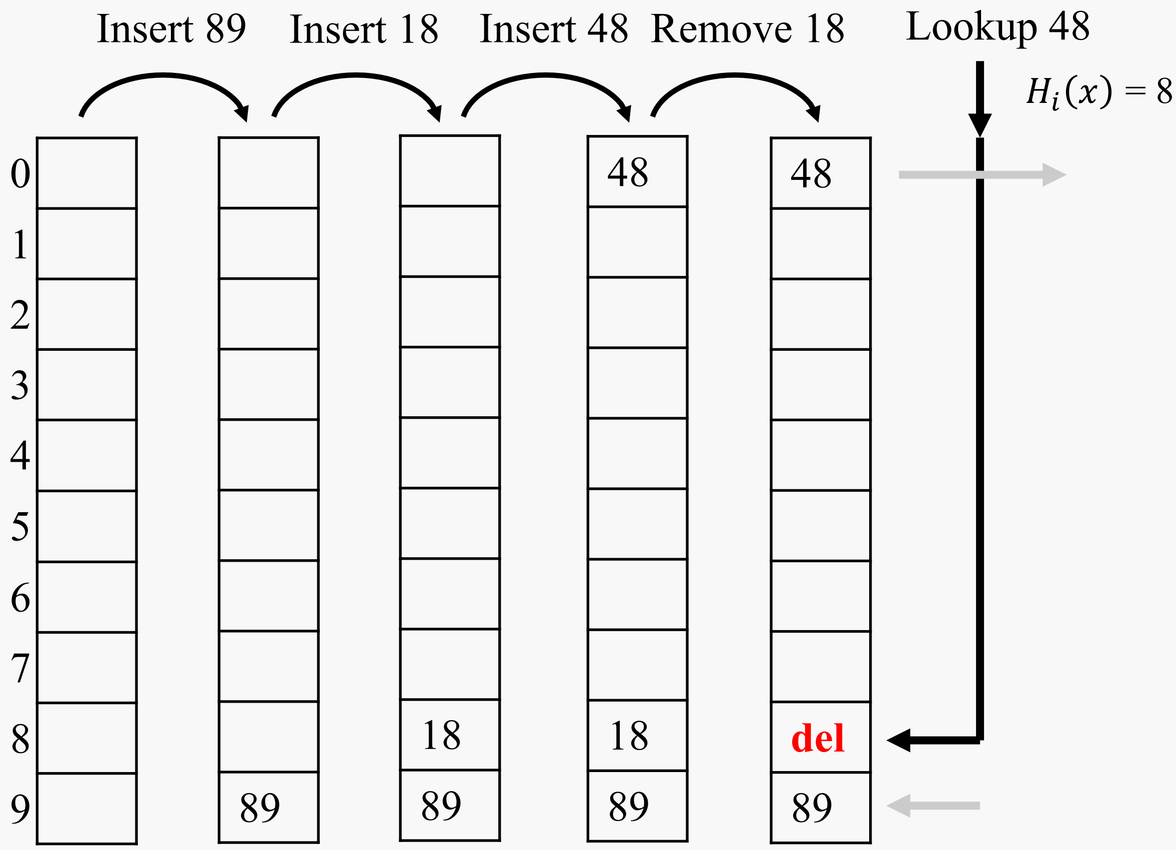
$H_i(x) = \ (x + i) \ \% \ 10$

Collision Resolution: Open Addressing

- **Open Addressing Removal**
  - Simple Removal clearly <u>prevents</u> finding elements in the table…
  - **Approach:** Lazy Removal
    - We mark the entry as “deleted” and thus “not empty”
    - Other Lookups will skip it, continuing to probe, eventually finding the correct key, if present in the table.

Collision Resolution: Open Addressing

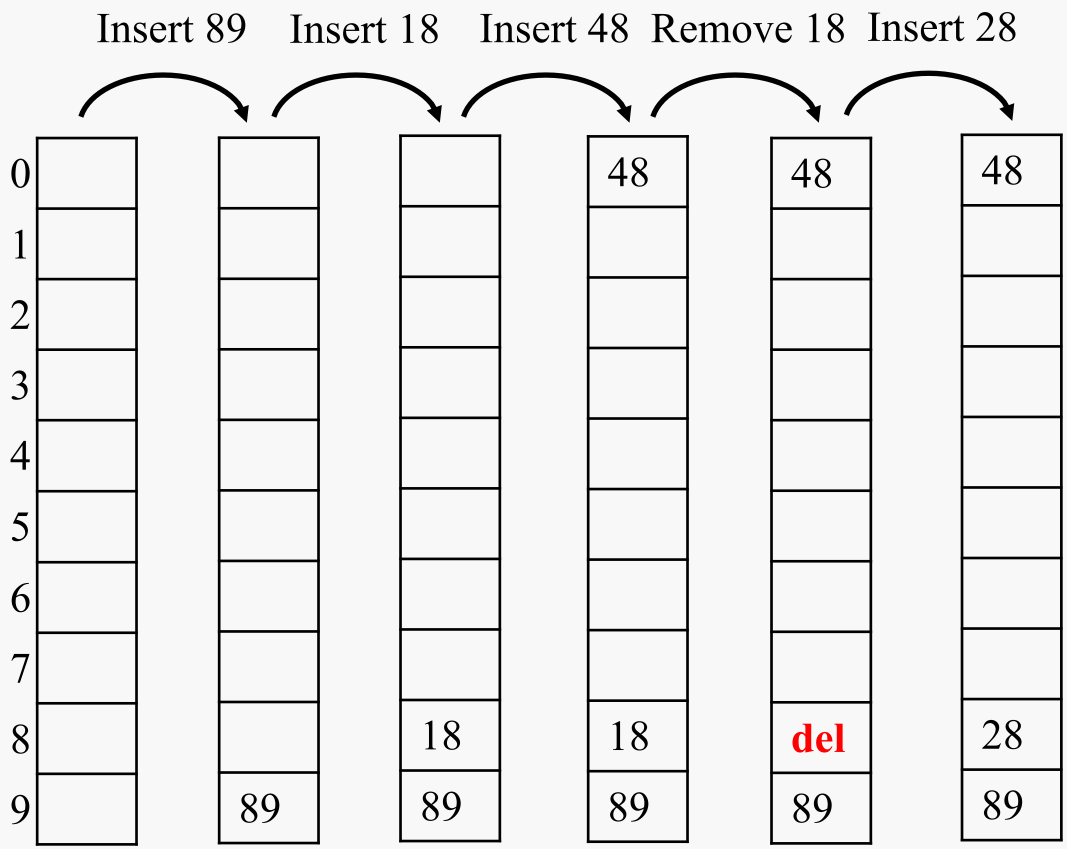
- **Open Addressing Lazy Removal**



$H_i(x) = \ (x + i)\ \%\ 10$

Collision Resolution: Open Addressing

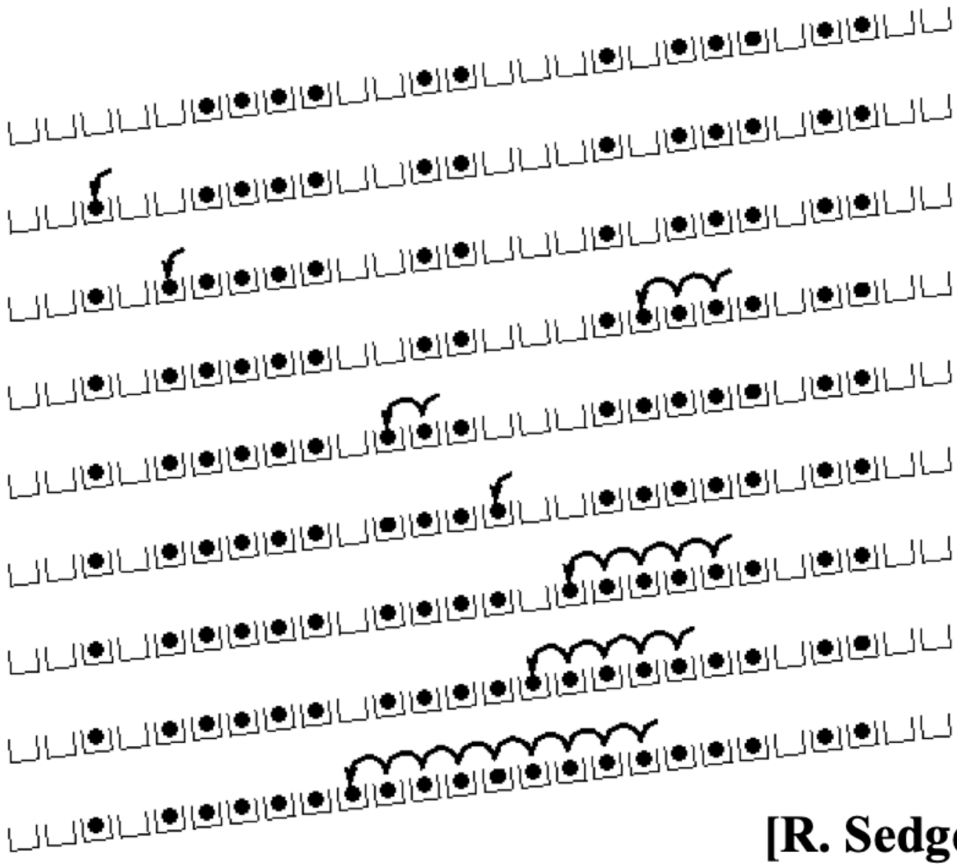
- **Open Addressing Lazy Removal**



$H_i(x) = (x + i)\ \%\ 10$

Collision Resolution: Open Addressing Issue

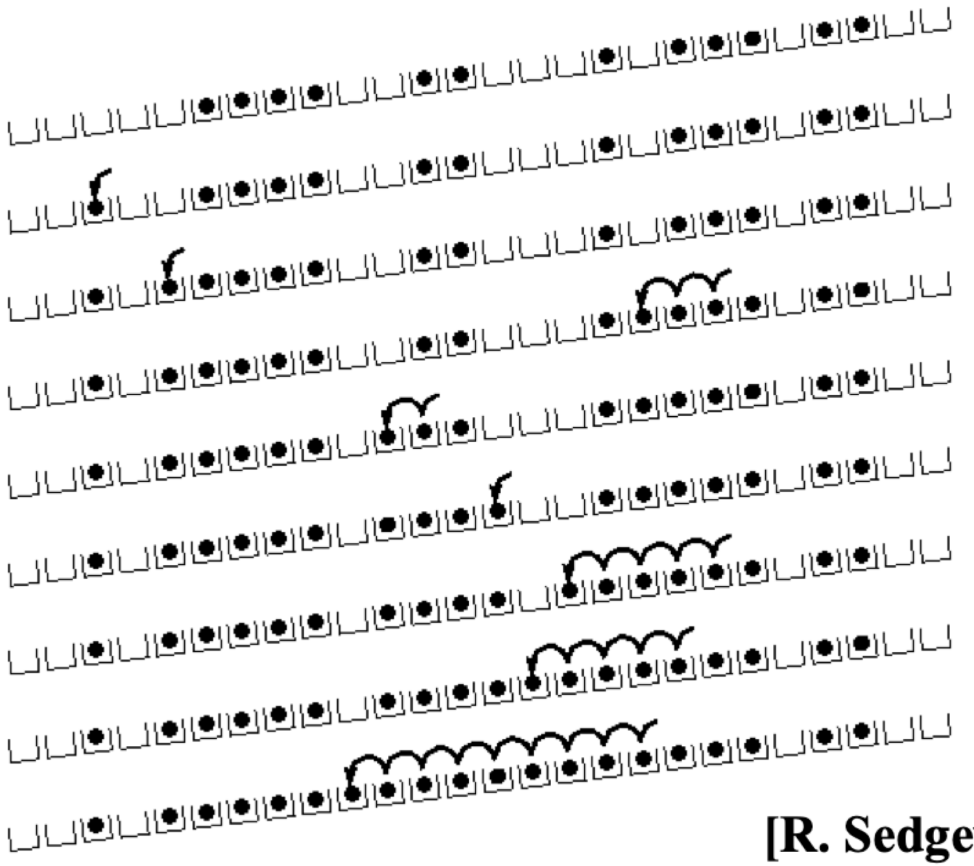
- Linear probing can lead to *(primary) clustering*



[R. Sedgewick]

Collision Resolution: Open Addressing Issue

- Linear Probing can lead to *(primary) Clustering*



[R. Sedgewick]

⇨ This means longer Search/Find Operations

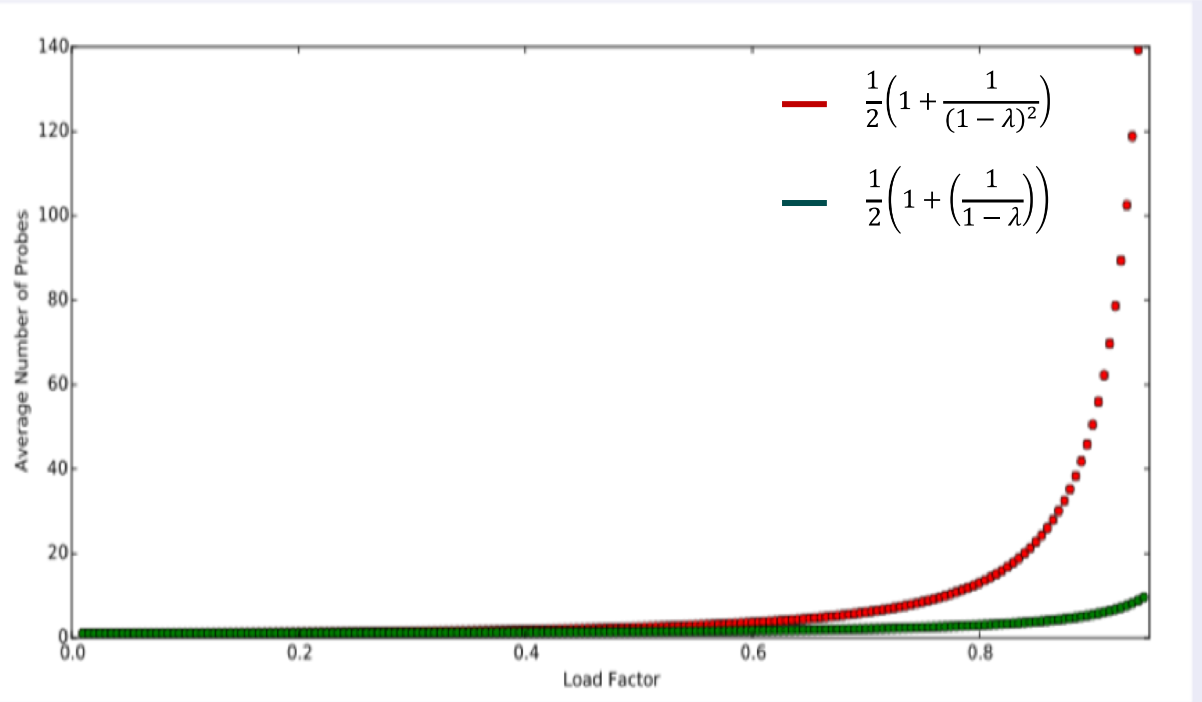
Collision Resolution: Open Addressing Performance

- Load Factor and Space Usage
  - Note that $\lambda \leq 1$, but eventually will be 1

- Average Number of Probes:
  - Ideal (*no clustering*)
    - insertion / unsuccessful search: $\left(\frac{1}{1-\lambda}\right)$
    - successful search: $\left(\frac{1}{\lambda}\right) log\left(1+\frac{1}{(1-\lambda)}\right)$
  - In Practice (*with clustering*)
    - insertion / unsuccessful search: $\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$
    - successful search: $\frac{1}{2}\left(1+\left(\frac{1}{1-\lambda}\right)\right)$

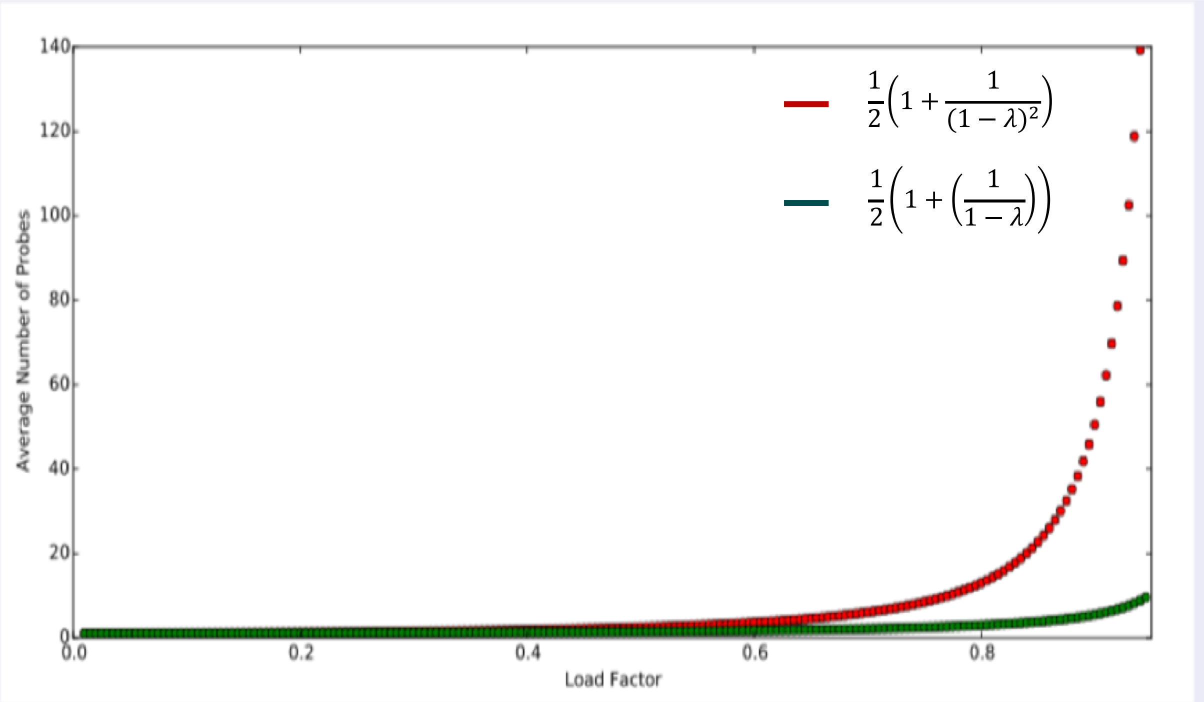
Collision Resolution: Open Addressing Performance

- **Observations:**
  - Performance Degrades Substantially whenever λ → 1
  - Solution: Need to make table large to control load factor



Collision Resolution: Open Addressing Performance

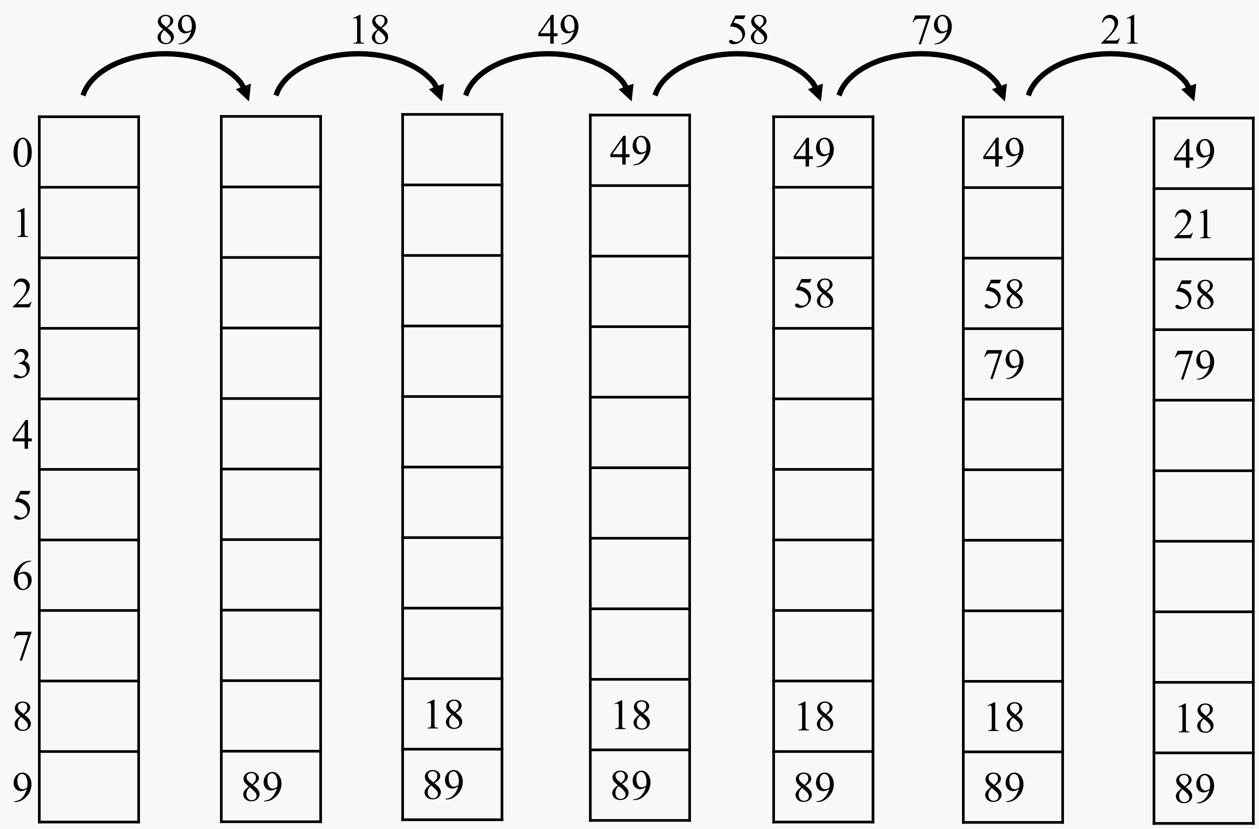
- **Observations:**
  - Performance Degrades Substantially whenever λ → 1
  - Solution: Need to make table large to control load factor



Quadratic probing ***eliminates*** *the issue of* ***primary clustering***

Collision Resolution: Open Addressing

- **Quadratic Probing Example**



$$H_i(x) = (x + i^2) \% 10$$

Note: *in the example, table size =10 just for simplification (as this should be a prime number)*

Collision Resolution: Open Addressing

- **Quadratic Probing**
  - **Does not guarantee** that a free position will always be found for a given element.
  - For example:

T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]
48		5	55		40	76

    - For all i, $(5+i^2) \bmod 7 \in \{0,2,5,6\}$. The proof is by induction. This generalizes: For all c,k, $(c+i^2) \bmod k = (c+(i-k)^2) \bmod k$
    - So, quadratic probing doesn't always **fill the table**.

> Good News: When the table size is prime, and quadratic probing is used, it is always possible to insert an element if the table is not filled more than 50%

  - Performance approaches the ideal case without aggregation
  - Alternative positions in the quadratic probing can be calculated with just one multiplication: $H_i = (H_{i-1} + 2\times i - 1)\ \%\ tablesize$

Hash Table: A Possible Implementation

```
template <class T> class HashTable {
// ...
enum EntryType {ACTIVE, EMPTY, DELETED};
private:
   struct HashEntry {
      T element;
      EntryType info;
      HashEntry(const T& e = T(), EntryType i = EMPTY):
                              element(e), info(i) {}
   };

   vector<HashEntry> array;
   int currentSize;
   const T ITEM_NOT_FOUND;

   bool is Active(int currentPos) const;
   int findPos(const T & x) const;
   void rehash();
};
```

Hash Table: A Possible Implementation

Searching

```
template <class T> int HashTable<T>::findPos(const T& x) const {
    int collisionNum = 0;
    int currentPos = hash(x, array.size());
    while(array[currentPos].info != EMPTY &&
                    array[currentPos].element != x) {
        currentPos += 2 * ++collisionNum - 1;
        if (currentPos >= array.size())
            currentPos -= array.size();
    }
    return currentPos;
}
```

collision

```
template <class T> const T& HashTable<T>::find(const T& x) const {
    int currentPos = findPos(x);
    if ( isActive(currentPos) )
      return array[currentPos].element;
    else
      return ITEM_NOT_FOUND;
}
```

Hash Table: A Possible Implementation

Insert

```
template <class T> void HashTable<T>::insert(const T& x){
    int currentPos = findPos(x);
    if ( isActive(currentPos) ) return;
    array[currentPos] = HashEntry(x, ACTIVE);
    if ( ++currentSize > array.size()/2 ) rehash();
}
```

rehash may be required

```
template <class T> void HashTable<T>::rehash(){
    vector<HashEntry> oldArray = array;
    array.resize(nextPrime(2 * oldArray.size()));
    for( int j = 0; j < array.size(); j++ )
        array[j].info = EMPTY;
    currentSize = 0;
    for( int i = 0; i < oldArray.size(); i++ )
        if ( oldArray[i].info == ACTIVE )
            insert(oldArray[i].element);
}
```

Hash Table: A Possible Implementation

Removal

```
template <class T> void HashTable<T>::remove(const T& x){
   int currentPos = findPos(x);
   if (isActive(currentPos) )
      array[currentPos].info = DELETED;
}
```

does not "eliminate" element from table

```
template <class T> bool HashTable<T>::isActive(int currentPos) const{
   return (array[currentPos].info == ACTIVE);
}
```

Class *unordered_set* (STL)

- class ***unordered_set*** in STL:

  unordered_set<T, HashFunc, EqualFunc>

- Some Methods
  - pair<iterator,bool> insert(const T& x)
    - return value:
      - iterator to the inserted element (or to the element that prevented the insertion)
      - bool: whether the insertion took place.
  - iterator erase(iterator it)
    - return value: iterator following the last removed element
  - iterator find(const T& x) const
  - iterator begin()
  - iterator end()
  - bool empty() const
  - void clear()

en.cppreference.com/w/cpp/container/unordered_set

Hash Table: A Use Case

- Count The Number of Occurrences of Words in a Text

Write a program that reads a text file and determines the list of words in it and the respective number of occurrences

- Use a Hash Table, that keeps the different words, and associates with each a counter
- For each Word, check if it already exists in the table
  - if it does not exist, insert it in Hash Table with counter = 1
  - if it exists, increment the corresponding counter (may need to eliminate the element from the table, and then insert it with the updated count).

Hash Table: A Use Case

```
class Text {
   ifstream f;
public:
   Text(string namef);
   string getWord();
   bool endText();
   ~Text() { f.close(); }
};
```

```
Text::Text(string namef) {
   f.open(namef.c_str());
   if (!f)
      throw FileNotFound();
}
```

```
string Text::getWord(){
   string w="";
   if (!f.eof())
      f>>pal;
   return w;
}
```

Hash Table: A Use Case

```
class WordFreq {
   string word;
   int frequency;
public:
   WordFreq(): word(""), frequency(0) {};
   WordFreq(string w) : word(w), frequency(1) {};
   string getWord() const { return word; }
   void incFrequency() { frequency ++; }
   // …
};
```

Hash Table: A Use Case

equality function

```
struct eqWF {
   bool operator() (const WordFreq& wf1, const WordFreq& wf2) const {
      return wf1.getWord()==wf2.getWord();
   }
};
```

hash function

```
struct hWF {
   int operator() (const WordFreq& wf) const  {
      string s1 = wf.getWord();
      int v = 0;
      for ( unsigned int i=0; i < s1.size(); i++ )
            v = 37*v + s1[i];
      return v;
   }
};
```

Hash Table: A Use Case

```
typedef unordered_set<WordFreq,hWF,eqWF>::iterator iteratorH;
typedef unordered_set<WordFreq,hWF,eqWF> tabH;

int main()  {
   Text tx("text1.txt");
   tabH tab1;
   while (!tx.endText()) {
      WordFreq wordf1 = WordFreq(tx.getWord());
      pair<iteratorH, bool> res = tab1.insert(wordf1);
      if ( res.second == false) {    //not inserted, already existed
         iteratorH it= res.first;
         WordFreq wordf = *it;
         tab1.erase(it);
         wordf.incFrequency();
         tab1.insert(wordf);
      }
   }
```

Hash Table: A Use Case

```
    cout << “words found:" << tab1.size() << endl;

    iteratorH it = tab1.begin();
    while (it != tab1.end()){
       cout << *it;
       it++;
    }
}
```

Hash Table Summary

- **Hash Tables: one of the Most Important Data Structures**
  - Efficient `find`, `insert`, and `delete`
  - Useful in many, many Real-world Applications

- **Important to use Good Hash Function**
  - Good distribution, uses enough of Keys Values
  - Not overly Expensive to Compute (bit shifts are good!)

- **Important to keep Hash Table at a good Size**
  - Prime Size
  - λ depends on Type of Table