# Hashing

Unit 3: Lecture 02

# Set Representation by Dictionary/Hashing

A dictionary may be implemented by using a **hash function** where each element of the dictionary is mapped into positions in a **hash table**

If the set member has key value x, it is **stored in the position f(x)**: f() is the hash function

In order to search a member with key value x in the hash table, simply f(x) is estimated and the **position f(x) is checked**: a match states a successful search or a failure otherwise

**How to choose** a hash function?

# Hash Function

When the set members are **numbers**, the key values are the numbers themselves however **for non-numeric members**, the key values are to be **estimated (how?)**

For character strings, the key value may be **estimated by using some character coding** (converting the strings into numbers)

When the key range is too large, a table is used with number of positions less than the key range: each position is called a **bucket**

For numeric keys (obtained directly or by coding), the **simplest hash function** is to use a modulo operator:

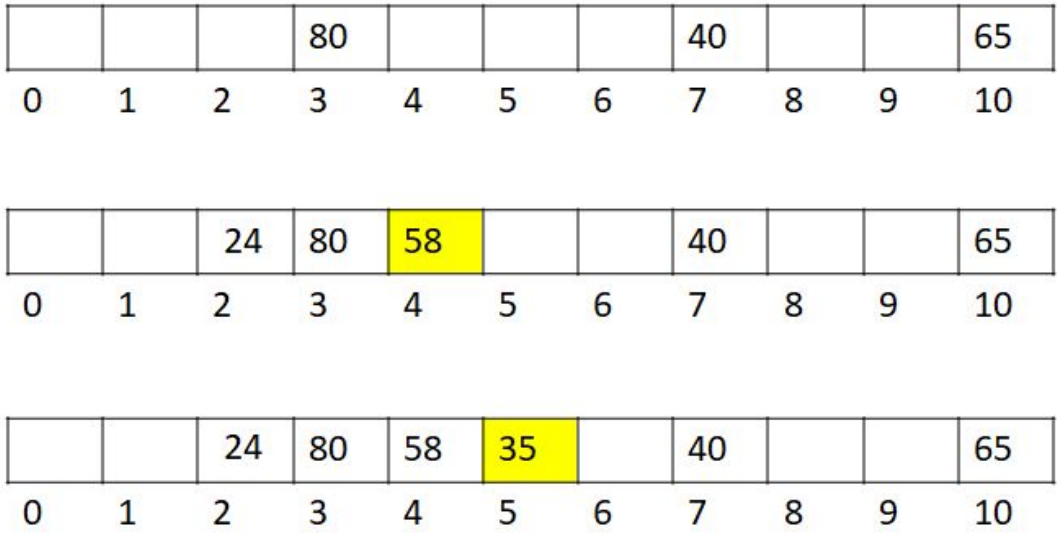
$$f(x)=x \bmod N$$

here N is the number of buckets (size of the hash table)

# Hash Function (contd.)

Although modulo operator is a simple operator, it results is **collision** (mapping more than one keys to same position)

Example:



# Hash Function (contd.)

After inserting 80, 40, 65 (initially) followed by 24 successfully, when 58 is to be inserted at position 3 (58 mod 11) a **collision occurs**

If each bucket is capable to store multiple members (**how?**), a collision may be handled by filling the bucket until it is full- a **full bucket** ultimately results in **an overflow**

As a simple solution to the collision, the key is placed at next available position (called **linear probing**) and thus the key 58 is placed at position 4 (the available position)

# Hash Function (contd.)

Similarly, the key 35 expects the position 2 (35 mod 11) which is already filled and thus the key is placed at next available position, i.e., 5 (next positions 3 and 4 are also already filled)

When using the hash function f(x)=x mod N, the **divisor N must be neither even** nor divisible by small odd numbers (why?)

If N is even, f(x) maps all even keys to even positions and odd keys to odd positions

The **ideal choice** for N is to be a **prime number** however, if there is no prime number closer to the size of the table, it is suggested to choose N in such a manner that it is not divisible by any number **between 2 and 19** (a guideline)

# Searching a Hash Table

In the hash table with linear probing:

Searching starts with the position mapped by the hash function and continues to examine successive buckets in the table treating the table as circular until

- a bucket containing the key is reached (indicates a successful search) **OR**
- an empty bucket is reached **OR**
- the initial key position is encountered without finding the key in the table

The later 2 cases indicate a failure (an unsuccessful search)

# Deletion from the Table

In a table with linear probing, **direct deletion** is not allowed as it may result an unstable table

Deletion must leave behind a table on which **search operation works efficiently** accordingly, the deletion may require to **move several keys**

The **search for the keys to move** begins just after the bucket vacated by the deleted key and proceeds to successive buckets until

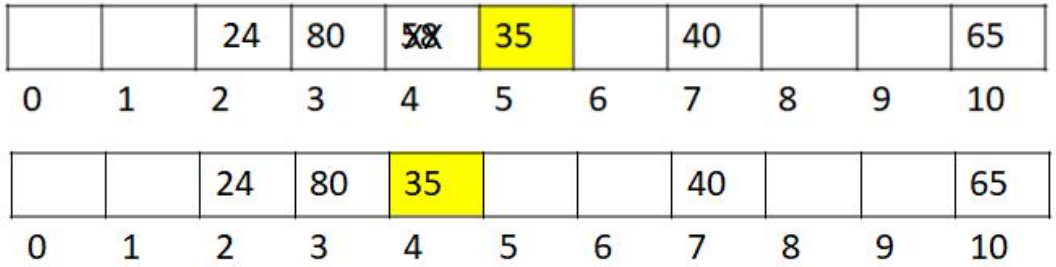
- either the empty bucket is encountered **OR**
- the bucket from which deletion took place is reached again (circularly)

When the keys are moved up, it is always taken care not to move any key **blindly** (how?)

# Deletion from the Table (contd.)

A blind deletion (without matching for the position) would cause a **failure of search** for such a key in future

**Example:** In the given example, deletion of 58 must be followed by moving up 35 otherwise the search for 35 would always be a failure




There are alternatives of such complex deletion operations (explore…)

# Performance of Linear Probing

The time to initialize the table of size N is O(N)

The worst case insert and search time is Θ(n) where n is the number of keys present in the table (the worst case occurs when all n keys are mapped to the same position)

# Hashing with Chains

The **alternative** to the linear probing is using a **chain: a linear list** attached with each bucket (of infinite length!)

Such an arrangement will handle the collisions **without overflow** (theoretically)

To **search** for a key, the position is obtained by division function f(x) followed by the search through the chain

To **insert** a key, first it is verified that the element is not already existing in the chain of appropriate position

To **delete** a key, the chain is searched for the key and deleted directly

# Exercise

Implement hash table with chaining and estimate the time complexity of search operation

# Reference Book

Data Structures, Algorithms and Applications in C++, Sartaj Sahni (2nd Ed.), 2008 University Press Hyderabad.