# CSE 4303
# Data Structures

## Topic: Hashing

Sabbir Ahmed
Assistant Prof | CSE | IUT
sabbirahmed@iut-dhaka.edu

# Hash-tables

- How to store this table?

| Name | Student ID |
|---|---|
| Student1 | 101 |
| Student2 | 102 |
| Student3 | 103 |
| Student4 | 104 |

# Hash-tables

- How to store this table?
  - Structures?

| Name | Student ID |
|---|---|
| Student1 | 101 |
| Student2 | 102 |
| Student3 | 103 |
| Student4 | 104 |

# Hash-tables

- How to store this table?
  - Structures?
  - {key, value} pair?

| Name | Student ID |
|---|---|
| Student1 | 101 |
| Student2 | 102 |
| Student3 | 103 |
| Student4 | 104 |

# Hash-tables

- How to store this table?
  - Structures?
  - {key, value} pair?
  - Python has ‘Dictionary’!
  - Heard about Hash-maps in C++?

| Name | Student ID |
|---|---|
| Student1 | 101 |
| Student2 | 102 |
| Student3 | 103 |
| Student4 | 104 |

# Hash-tables

| Name | Student ID |
|---|---|
| Student1 | 101 |
| Student2 | 102 |
| Student3 | 103 |
| Student4 | 104 |

- How to store this table?
  - Structures?
  - {key, value} pair?
  - Python has ‘Dictionary’!
  - Heard about Hash-maps in C++?

- **Hash Tables**: values stored as {key, value(s)} pair.
  - ‘keys’ used to access value(s) (~names in this example)
  - Keys work as array-index!

# Hash-tables

| Name | Student ID |
|---|---|
| Student1 | 101 |
| Student2 | 102 |
| Student3 | 103 |
| Student4 | 104 |

- How to store this table?
  - Structures?
  - {key, value} pair?
  - Python has ‘Dictionary’!
  - Heard about Hash-maps in C++?

- **Hash Tables**: values stored as {key, value(s)} pair.
  - ‘keys’ used to access value(s) (~names in this example)
  - Keys work as array-index!

- **Hashing** can access items in O(1) time.
- This is only possible using ‘Hash-tables’

# Hashing

- Hashing is a technique used to implement hash tables.

- Great technique to perform
  - Insertions
  - Deletions
  - Searching

  } O(1)

# Hashing

- Hashing is a technique used to implement hash tables.

- Great technique to perform
  - Insertions
  - Deletions
  - Searching

  } O(1)

- Not efficient in operations like
  - findMax
  - findMin
  - Print data (the entire table) in sorted order

  } Not great when ordering is required

# General Idea

- Hash Table is merely an array of fixed size containing the items.
  - size of the array/table as $TableSize$.
  - Items stored in the table are indexed by $0$ … $TableSize - 1$.

# General Idea

- Hash Table is merely an array of fixed size containing the items.
  - size of the array/table as $TableSize$.
  - Items stored in the table are indexed by $0$ … $TableSize - 1$.
- Key is used to find the index of the associated item in the table.

# General Idea

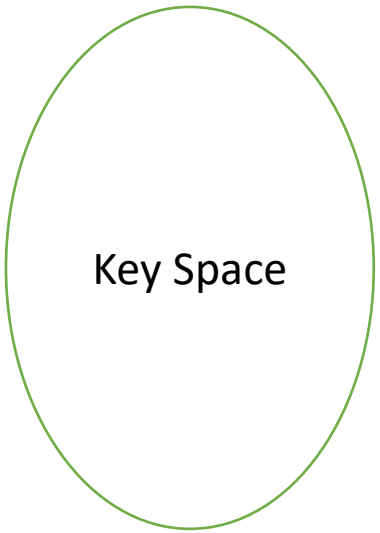
- Hash Table is merely an array of fixed size containing the items.
  - size of the array/table as $TableSize$.
  - Items stored in the table are indexed by $0$ … $TableSize - 1$.
- Key is used to find the index of the associated item in the table.
  - Keys are mapped to numbers (i.e. index) ranging from $0 \ldots TableSize - 1$.

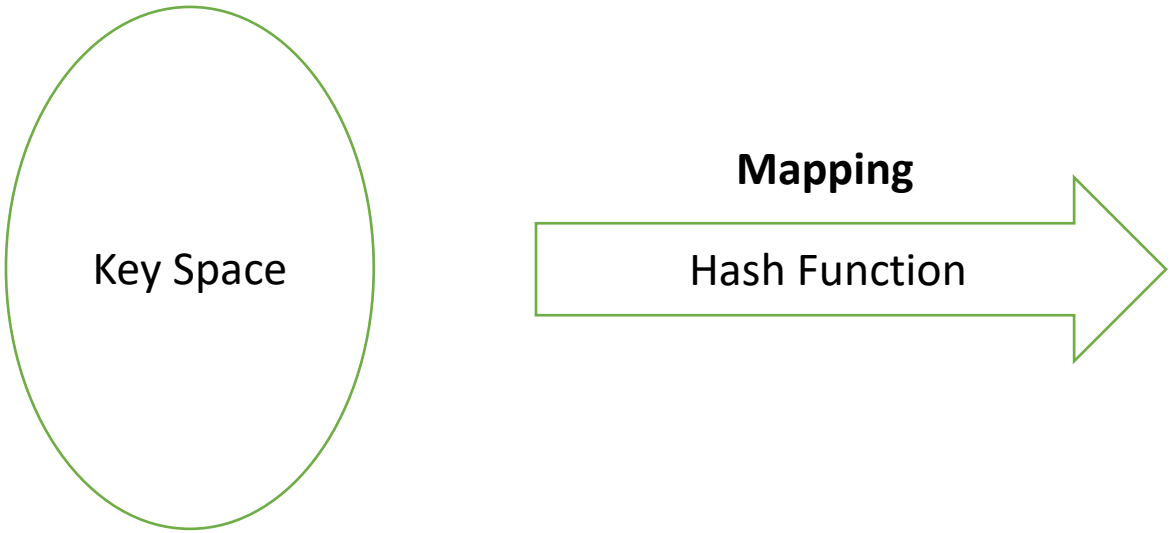
# General Idea

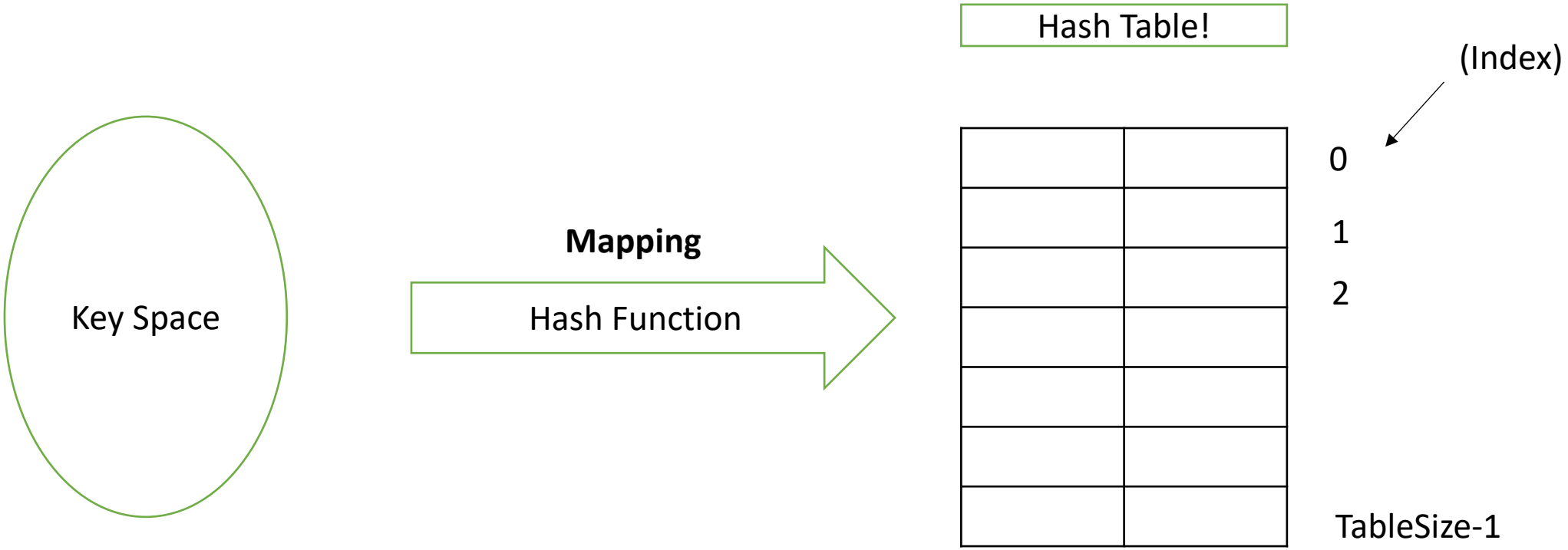
- Hash Table is merely an array of fixed size containing the items.
  - size of the array/table as $TableSize$.
  - Items stored in the table are indexed by $0$ … $TableSize - 1$.
- Key is used to find the index of the associated item in the table.
  - Keys are mapped to numbers (i.e. index) ranging from $0 \ldots TableSize - 1$.
  - Key can be an integer/string/value of any type.
  - E.g. a Name/ID that is part of a large student database.

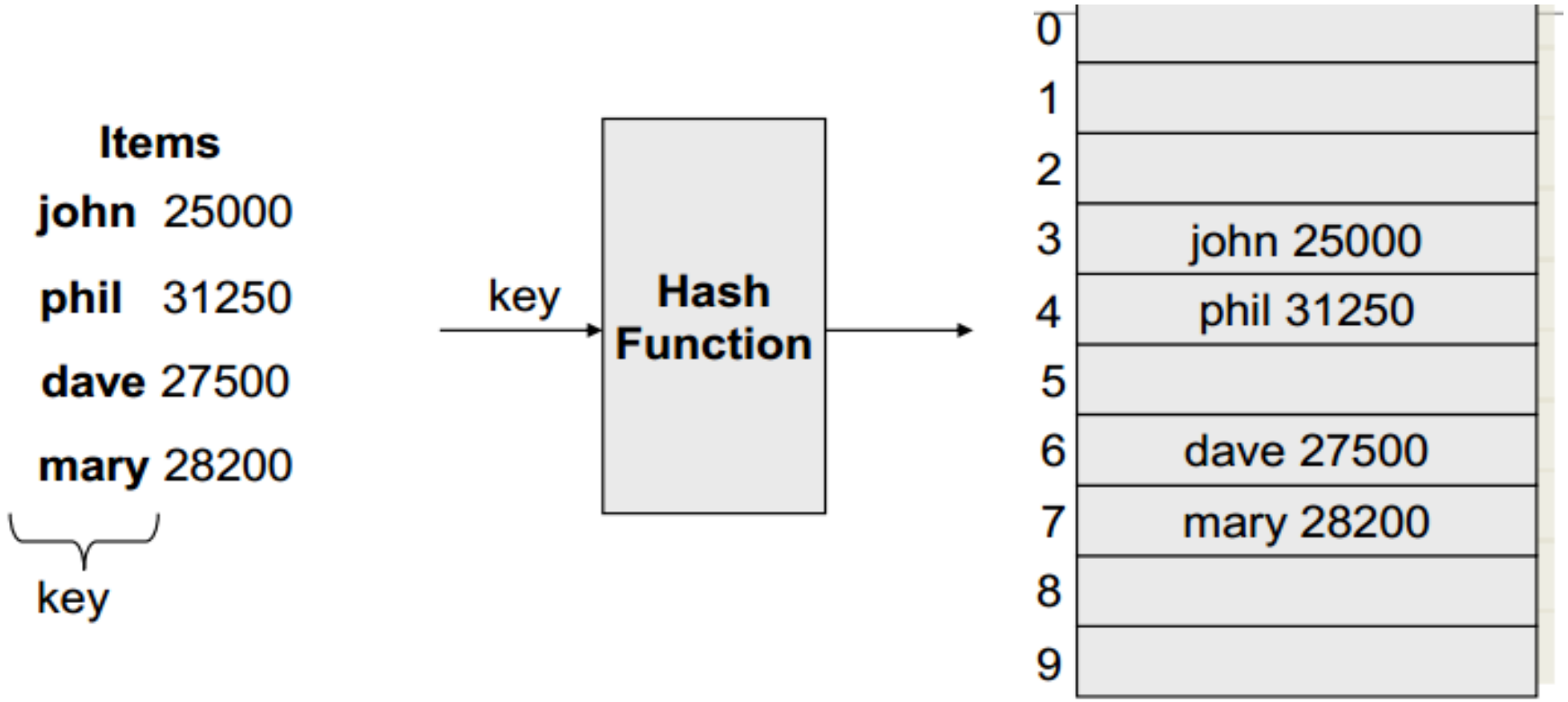
# General Idea

- Hash Table is merely an array of fixed size containing the items.
  - size of the array/table as $TableSize$.
  - Items stored in the table are indexed by $0$ … $TableSize - 1$.
- Key is used to find the index of the associated item in the table.
  - Keys are mapped to numbers (i.e. index) ranging from $0 \ldots TableSize - 1$.
  - Key can be an integer/string/value of any type.
  - E.g. a Name/ID that is part of a large student database.
- The mapping done using Hash Function.


Key Space


Key Space
Mapping
Hash Function


Hash Table!
(Index)
Key Space
Mapping
Hash Function
0
1
2
TableSize-1


Items
john 25000
phil 31250
dave 27500
mary 28200
key
key
Hash Function
0
1
2
3
john 25000
4
phil 31250
5
6
dave 27500
7
mary 28200
8
9

# Hash Functions

- Maps key to the Hash-table.
- Requirements:

# Hash Functions

- Maps key to the Hash-table.
- Requirements:
  - Simple to compute (must satisfy O(1)! )

# Hash Functions

- Maps key to the Hash-table.
- Requirements:
  - Simple to compute (must satisfy O(1)! )
  - Distribute the keys evenly among the cells

# Hash Functions

- Maps key to the Hash-table.
- Requirements:
  - Simple to compute (must satisfy O(1)! )
  - Distribute the keys evenly among the cells
  - Should minimize collisions (happens when multiple keys are mapped to the same index).

# Hash Functions

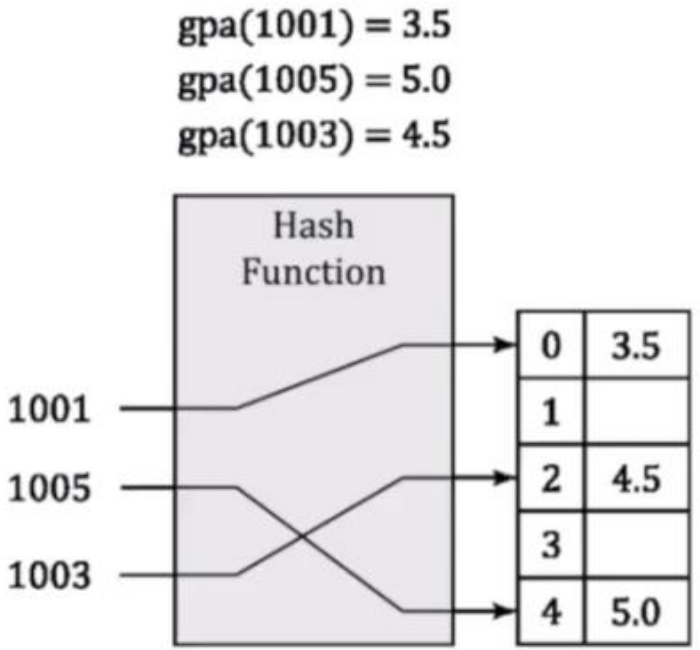
- Maps key to the Hash-table.
- Requirements:
  - Simple to compute (must satisfy O(1)! )
  - Distribute the keys evenly among the cells
  - Should minimize collisions (happens when multiple keys are mapped to the same index).

- If we know which keys will occur (in advance) we can design perfect hash functions, but we don’t know!
- Need to design hash functions to properly distribute the items in the hash tables.

# Designing Hash functions

- If keys are integers, $(key \ \% \ TableSize)$ is a general strategy.

```
index = hash_function(key)

Hash_function (key){
    index = key % TableSize
    return index -1;
}
```

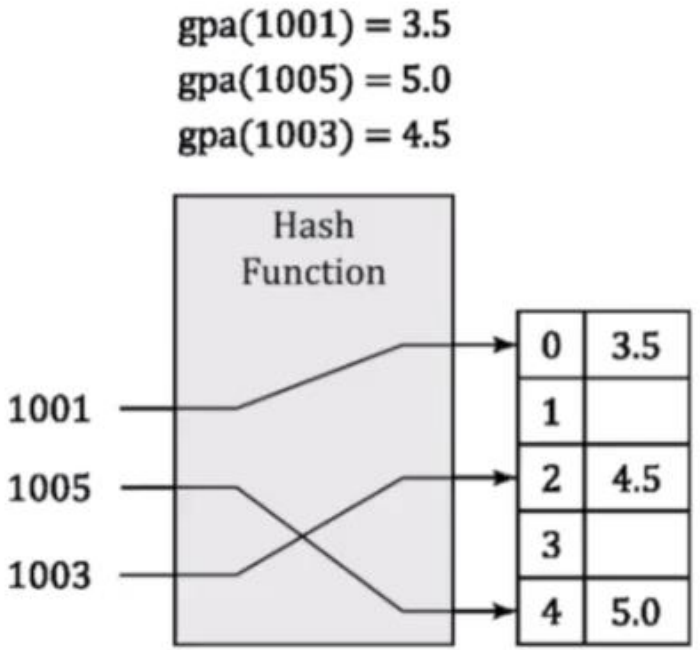


# Designing Hash functions

- If keys are integers, $(key \ \% \ TableSize)$ is a general strategy.

- If the keys are strings, the Hash function needs more care.
  - First needed to be converted into a numeric value.

```
index = hash_function(key)

Hash_function (key){
    index = key % TableSize
    return index -1;
}
```



# Hash Functions

- Challenges:
  - Keys may not be numeric.

# Hash Functions

- Challenges:
  - Keys may not be numeric.
  - The number of possible keys are much larger than the space available in the table.

# Hash Functions

- Challenges:
  - Keys may not be numeric.
  - The number of possible keys are much larger than the space available in the table.
  - Different keys may map into the same location (collision).
    - If there are too many collisions, the performance of hash-table

# More Ideas to design Hash functions

- Truncation:
  - Key = 123456789, map to a table off 1000 addresses by picking 3 digits of the key

# More Ideas to design Hash functions

- Truncation:
  - Key = 123456789, map to a table off 1000 addresses by picking 3 digits of the key
- Folding:
  - 123|456|789 : add the folds and take the mod. = 1368%1000=368

# More Ideas to design Hash functions

- Truncation:
  - Key = 123456789, map to a table off 1000 addresses by picking 3 digits of the key
- Folding:
  - 123|456|789 : add the folds and take the mod. = 1368%1000=368
- Key mod N:
  - N is the size of the table. Better if N is prime (helps in collision handling).

# More Ideas to design Hash functions

- Truncation:
  - Key = 123456789, map to a table off 1000 addresses by picking 3 digits of the key
- Folding:
  - 123|456|789 : add the folds and take the mod. = 1368%1000=368
- Key mod N:
  - N is the size of the table. Better if N is prime (helps in collision handling).
- Squaring:
  - Square the key and then truncate.

# More Ideas to design Hash functions

- Add up the ASCII values of all characters of the key.

```
int hash(const string &key, int tableSize)
{
        int hasVal = 0;

        for (int i = 0; i < key.length(); i++)
                hashVal += key[i];
        return hashVal % tableSize;
}
```

- Simple to implement and fast.

# More Ideas to design Hash functions

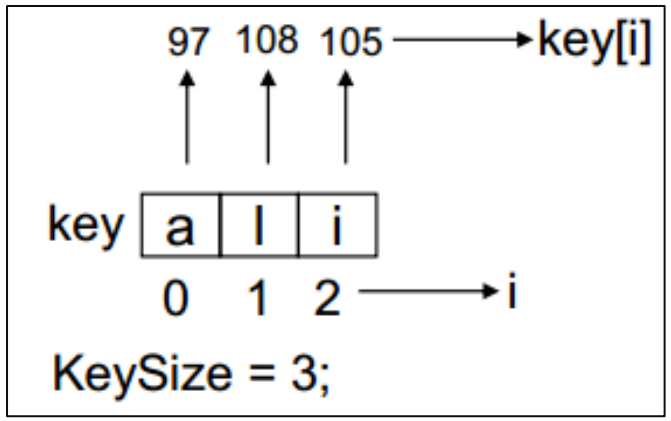
- Add up the ASCII values of all characters of the key.

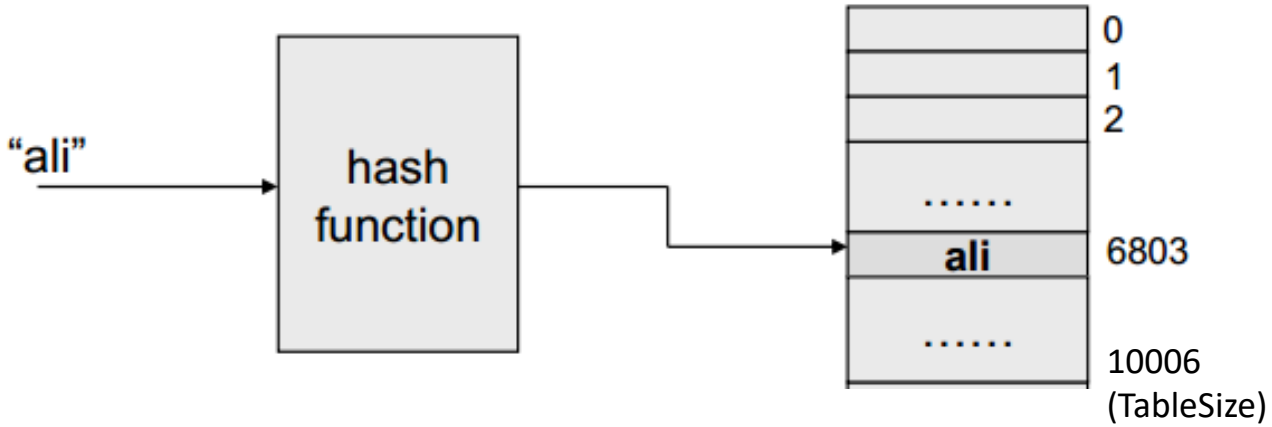
```
int hash(const string &key, int tableSize)
{
        int hasVal = 0;

        for (int i = 0; i < key.length(); i++)
                hashVal += key[i];
        return hashVal % tableSize;
}
```

- Simple to implement and fast.

- However, if the TableSize is large, the function does not distribute keys well.
- E.g. TableSize = 10000, key_length <=8, Hash function can assume values only between 0 and 1016.

$$\text{Hash(key)} = \sum_{i=0}^{keySize-1} key[keySize - i - 1] \times 37^i$$




$$Hash('ali') = 105 \times 37^0 + 108 \times 37^1 + 97 \times 37^2 = 6803$$

# Collision resolution

- Collision: Trying to insert an element that hashes to the same value as an already inserted element.

# Collision resolution

- Collision: Trying to insert an element that hashes to the same value as an already inserted element.

- Several methods to deal with collisions:
  - Separate Chaining/ Open Hashing
  - Open addressing/ Closed Hashing
    - Linear Probing
    - Quadratic Probing
    - Double Hashing

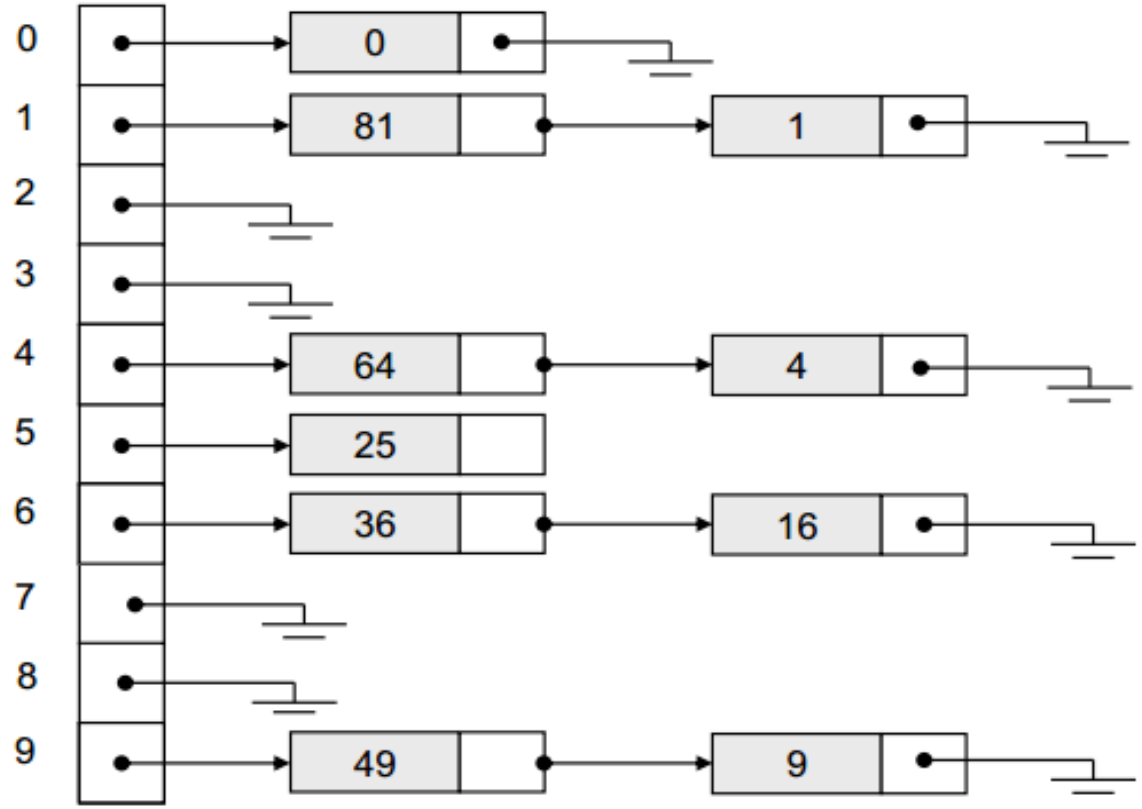
# Separate chaining

- Idea: maintain individual lists for all elements that hash to same value.

# Separate chaining

hash(key) = key % 10.

Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81



# Separate chaining

- Idea: maintain individual lists for all elements that hash to same value.
  - Array elements of Hash table are pointers to the first nodes of the lists.

# Separate chaining

- Idea: maintain individual lists for all elements that hash to same value.
  - Array elements of Hash table are pointers to the first nodes of the lists.
  - Insertion takes place to the front of the list (same as inserting in beginning of Linked List).

# Separate chaining

- Idea: maintain individual lists for all elements that hash to same value.
  - Array elements of Hash table are pointers to the first nodes of the lists.
  - Insertion takes place to the front of the list (same as inserting in beginning of Linked List).

- Advantages:
  - Better space utilization for large items.
  - Simple collision handling: searching Linked List
  - Overflow: we can store more items than the hash TableSize.

# Separate Chaining (Operations)

- Initializations: all entries are set to NULL.

- Find:
  - Locate the cell using Hash function.
  - Perform sequential search on the Linked list in the cell.

# Separate Chaining (Operations)

- Initializations: all entries are set to NULL.

- Find:
  - Locate the cell using Hash function.
  - Perform sequential search on the Linked list in the cell.
- Insertion:
  - Locate the cell using Hash function.
  - Insert it as the first item in the list.

# Separate Chaining (Operations)

- Initializations: all entries are set to NULL.

- Find:
  - Locate the cell using Hash function.
  - Perform sequential search on the Linked list in the cell.
- Insertion:
  - Locate the cell using Hash function.
  - Insert it as the first item in the list.
- Deletion:
  - Locate the cell using Hash function
  - Delete the item from Linked list.

- Collisions are very likely.
- What is the average length of the lists?

- Load factor ($\lambda$) :
  - Ratio of the number of items (N) and TableSize.
  - i.e. $\lambda = N \ / \ TableSize$

# Separate chaining(cost of searching)

- COST = $O(1)$ to evaluate the Hash function +
  Time needed to traverse the List.

# Separate chaining(cost of searching)

- COST = $O(1)$ to evaluate the Hash function +
  Time needed to traverse the List.

- Unsuccessful search: $O(1 + \lambda)$
  - Need to traverse the entire list. Need to compare $\lambda$ nodes on average.

# Separate chaining(cost of searching)

- COST = $O(1)$ to evaluate the Hash function +
  Time needed to traverse the List.

- Unsuccessful search: $O(1 + \lambda)$
  - Need to traverse the entire list. Need to compare $\lambda$ nodes on average.

- Successful search:
  - On average, need to check half of the nodes while searching for a certain element.
  - Average cost for searching = $1 + \lambda / 2$

# Chaining (Pros and cons)

- Pros:
  - Simple to implement
  - Hash-table never fills up. Can always add more items.
  - Less sensitive to the hash function and load factor.
  - Widely used when it is unknown how many and how frequently keys may be inserted or deleted.

# Chaining (Pros and cons)

- Pros:
  - Simple to implement
  - Hash-table never fills up. Can always add more items.
  - Less sensitive to the hash function and load factor.
  - Widely used when it is unknown how many and how frequently keys may be inserted or deleted.
- Cons:
  - Wastage of space (some parts of the hash table may never be used.)
  - If the chain becomes long, search time becomes $O(n)$.
  - Use extra space for links.

# *Hashing: Open Addressing*

# Collision resolution with open addressing

- In Open addressing, all the items go inside the table.
  - No chain is maintained.

# Collision resolution with open addressing

- In Open addressing, all the items go inside the table.
  - No chain is maintained.

- Bigger table may be needed.
  - Generally Load factor ($\lambda$) should be $<= 0.5$.

# Collision resolution with open addressing

- In Open addressing, all the items go inside the table.
  - No chain is maintained.

- Bigger table may be needed.
  - Generally Load factor ($\lambda$) should be $<= 0.5$.

- Collision can still occur!
  - Alternative cells are tried until an empty cell is found.

# Open addressing

- More formally:
  - Cells $h_0(x)$, $h_1(x)$, $h_2(x)$ … are tried in succession where
    $$h_i(x) = \big(x + f(i)\big)\ mod\ TableSize, with\ f(0) = 0$$
  - The function $f(i)$ is the collision resolution strategy.
- Common collision resolution strategies:
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

# Linear Probing

- Collisions are handled by sequentially searching the table until an empty cell is found.
  - i.e. $f(i) = i$

# Linear Probing

- Collisions are handled by sequentially searching the table until an empty cell is found.
  - i.e. $f(i) = i$
- Example:
  - Insert items with keys 89, 18, 49, 58, 9 into an empty hash table.
  - Table size is 10.

  - Insert 89: $(89 + f(i))\ \%\ 10\ =\ (89 + 0)\ \%\ 10\ = 9$
  - Insert 59: $(59\ +\ f(0))\ \%10\ =\ 59 + 0\ \%10\ =\ 9$
    - i=1, $59\ ->\ (59\ +\ f(1))\ \%10\ =\ 59 + 1\ \%10\ =\ 0$

# Linear Probing

- Collisions are handled by sequentially searching the table until an empty cell is found.
  - i.e. $f(i) = i$
- Example:
  - Insert items with keys 89, 18, 49, 58, 9 into an empty hash table.
  - Table size is 10.

  - Insert 89: $(89 + f(i))\ \%\ 10\ =\ (89 + 0)\ \%\ 10\ = 9$
  - Insert 59: $(59\ +\ f(0))\ \%10\ =\ 59 + 0\ \%10\ =\ 9$
    - i=1, $59\ ->\ (59\ +\ f(1))\ \%10\ =\ 59 + 1\ \%10\ =\ 0$
  - Insert 69:
    - $69 +\ f(0)\ \%10\ =\ 9$
    - $69 +\ f(1)\ \%10\ =\ 0$
    - $69 +\ f(2)\ \%10\ =\ 1$

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

| | 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 |

**Figure 5.11** Hash table with linear probing, after each insertion

# Linear probing (clustering problem)

- As long as the table is big enough, a free cell can always be found.
  - Time to find that space can get quite large.

- Even if the table is relatively empty, blocks of occupied cells start forming.
  - This effect is called ‘primary clustering’.

# Linear probing (clustering problem)

- As long as the table is big enough, a free cell can always be found.
  - Time to find that space can get quite large.

- Even if the table is relatively empty, blocks of occupied cells start forming.
  - This effect is called ‘primary clustering’.

- Any key that hashes into the cluster will require several attempts to resolve the collision, and then it will be added to the cluster.

# Analysis of insertion, search

- Average number of cells examined in an insertion using linear probing $= (1 + 1/(1-\lambda)^2)/2$
  - (Proof is beyond the scope of this lecture.)

- An unsuccessful search costs the same as insertion.
- Cost of successful search = cost that was necessary to insert X.

- If $\lambda$=0.5, avg comparison for insertion is 2.5.

# Quadratic probing

- Eliminates the primary clustering problem of Linear probing.
- Collision function is quadratic.
- Popular choice is $f(i)$=$i$^2.

# Quadratic probing

- Eliminates the primary clustering problem of Linear probing.
- Collision function is quadratic.
- Popular choice is $f(i)$=$i$^2.

- If the hash function evaluates to ‘h’, and that cell is already occupied, we try cells $h$+1^2, $h$+2^2, $h$+3^2…$h$+$i$^2

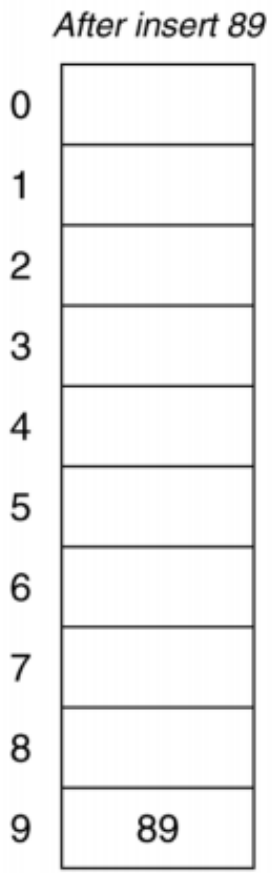
- i.e. it examines cells 1, 4, 9 and so on cells away from the original probe.

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

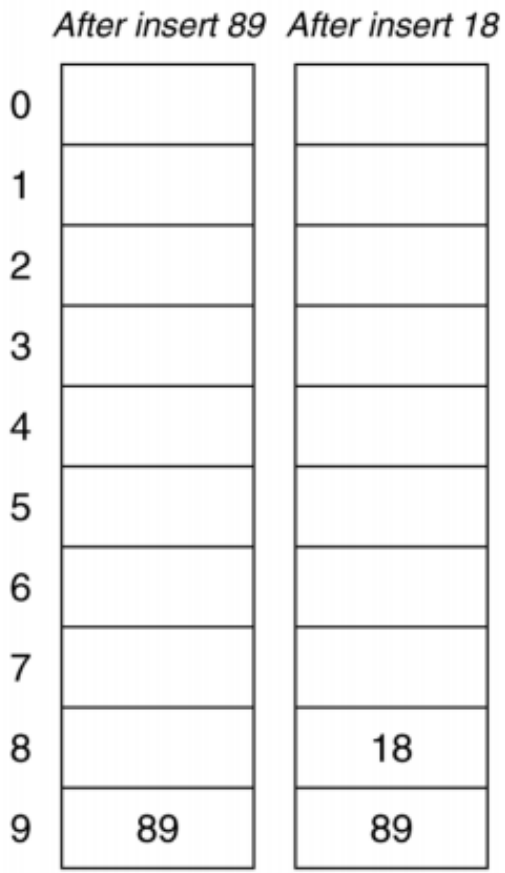
| | *After insert 89* | *After insert 18* | *After insert 49* | *After insert 58* | *After insert 9* |
|---|---|---|---|---|---|
| 0 | | | 49 | 49 | 49 |
| 1 | | | | | |
| 2 | | | | 58 | 58 |
| 3 | | | | | 9 |
| 4 | | | | | |
| 5 | | | | | |
| 6 | | | | | |
| 7 | | | | | |
| 8 | | 18 | 18 | 18 | 18 |
| 9 | 89 | 89 | 89 | 89 | 89 |

A quadratic probing hash table after each insertion.

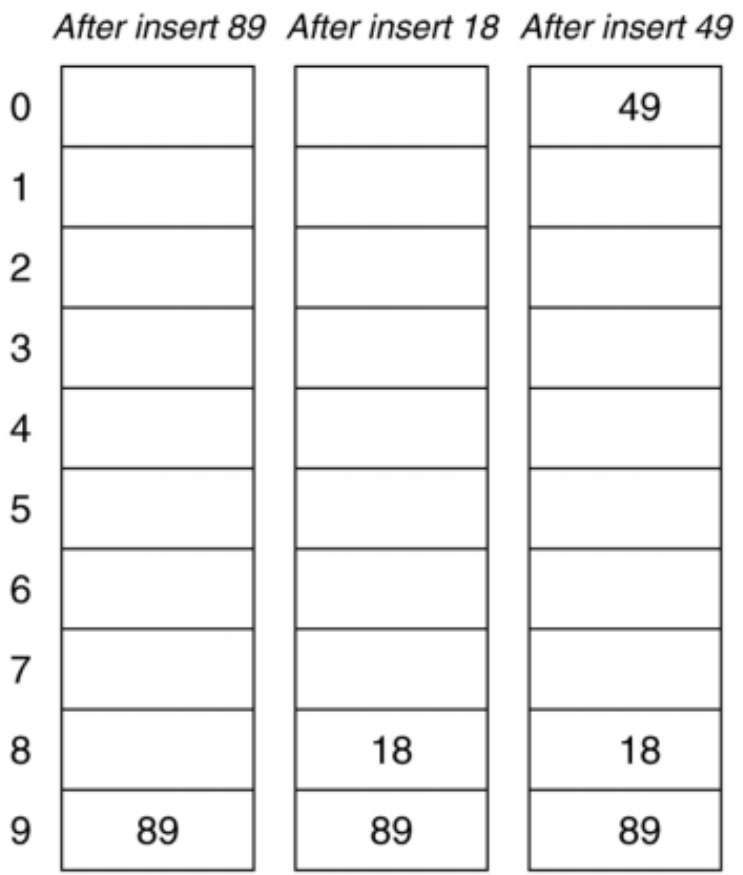
[Note: The table size is poorly chosen because it is not a prime number.]



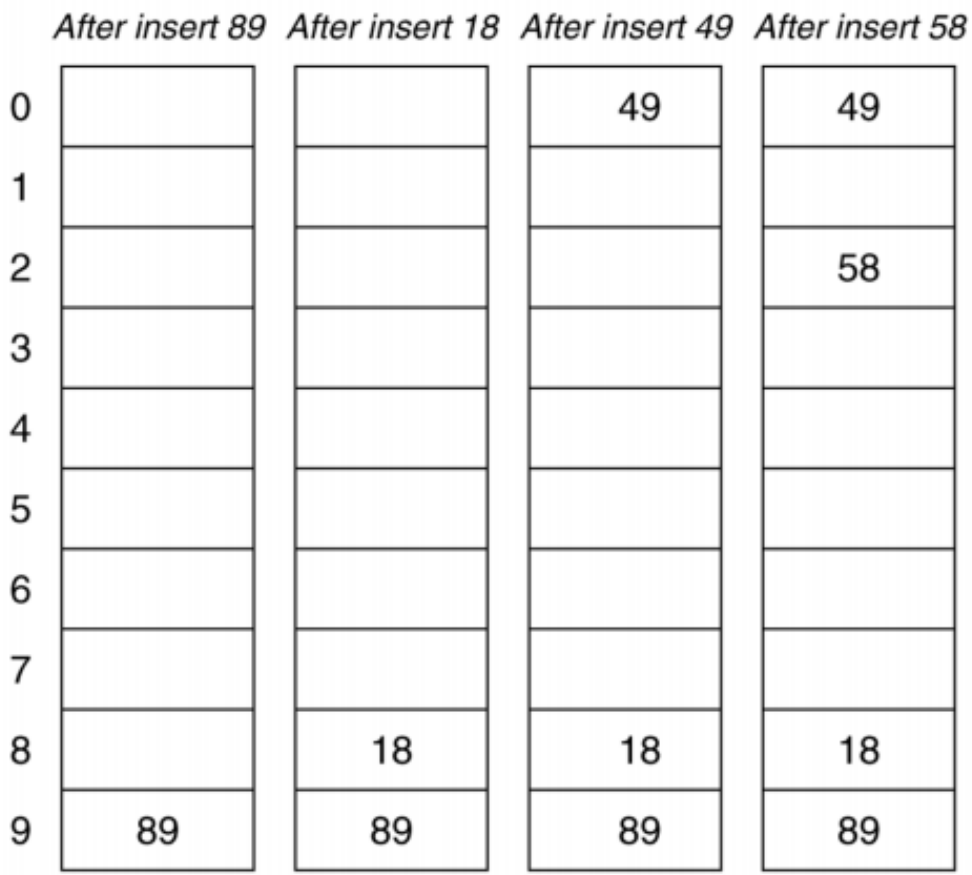

- Insert 89:
  - $hash_i(x) = \big(x + f(i)\big)\ mod\ TableSize,$
  - $f(i) = i^2,\ f(0) = 0\ , TableSize = 10$
  - $hash_0(89) = (89 + 0)\ \%\ 10 = 9$
  - No collision at index-9. So insert there.




- Insert 18:
  - $hash_0(18) = (18 + 0)\ \%\ 10 = 8$
  - No collision at index-8. So insert there.

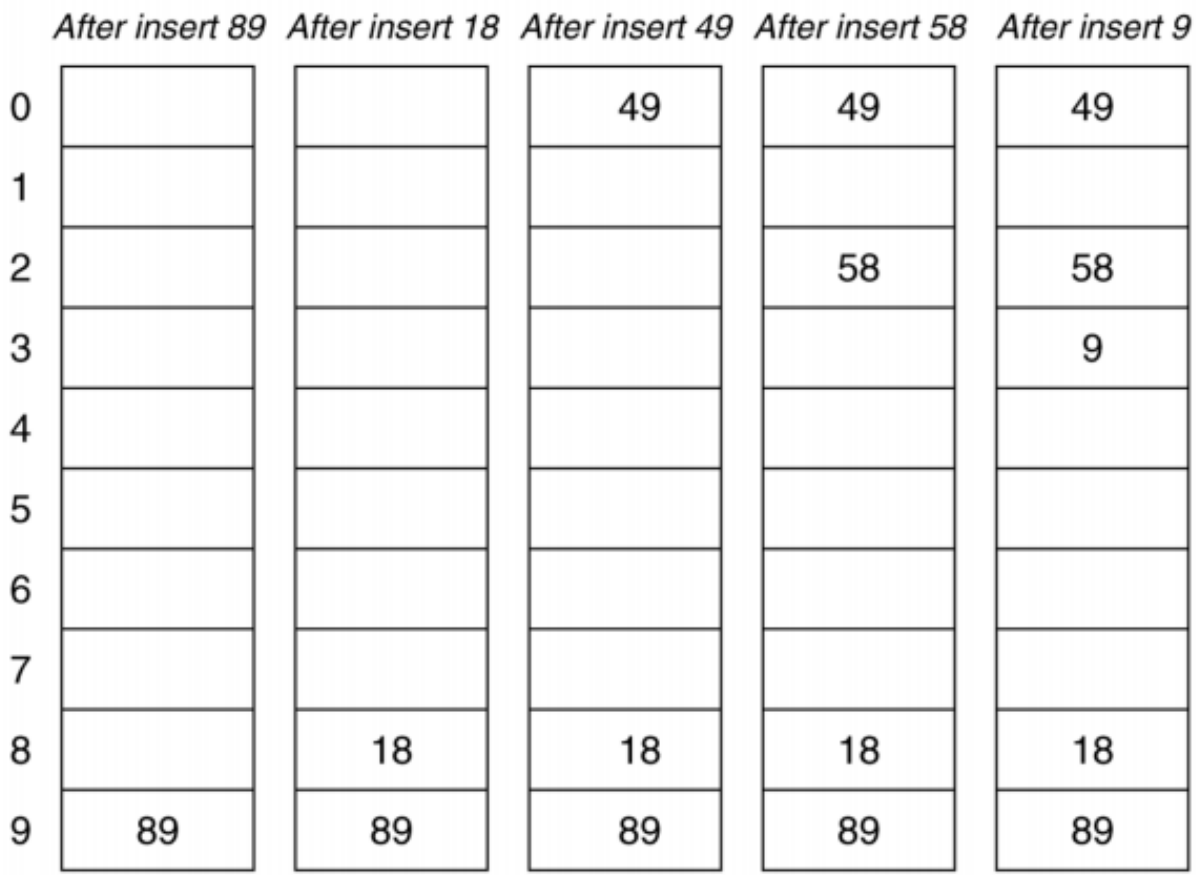



- Insert 49:
  - $hash_0(49) = (49 + 0)\ \%\ 10 = 9$
  - Collision at index-9.
  - Try $i = 1, f(1) = 1^2 = 1$
  - $hash_1(49) = (49 + 1)\ \%\ 10 = 0$
  - No collision at index-0. Insert there.



| | After insert 89 | After insert 18 | After insert 49 | After insert 58 |
|---|---|---|---|---|
| 0 | | | 49 | 49 |
| 1 | | | | |
| 2 | | | | 58 |
| 3 | | | | |
| 4 | | | | |
| 5 | | | | |
| 6 | | | | |
| 7 | | | | |
| 8 | | 18 | 18 | 18 |
| 9 | 89 | 89 | 89 | 89 |

- Insert 58:
  - $hash_0(58) = (58 + 0)\ \%\ 10 = 8$
  - Collision at index-8.
  - Try $i = 1, f(1) = 1^2 = 1$
  - $hash_1(58) = (58 + 1)\ \%\ 10 = 9$
  - Collision at index-9.
  - Try $i = 2, f(2) = 2^2 = 4$
  - $hash_2(58) = (58 + 4)\ \%\ 10 = 2$
  - No collision at index-2. Insert there.



| | After insert 89 | After insert 18 | After insert 49 | After insert 58 | After insert 9 |
|---|---|---|---|---|---|
| 0 | | | 49 | 49 | 49 |
| 1 | | | | | |
| 2 | | | | 58 | 58 |
| 3 | | | | | 9 |
| 4 | | | | | |
| 5 | | | | | |
| 6 | | | | | |
| 7 | | | | | |
| 8 | | 18 | 18 | 18 | 18 |
| 9 | 89 | 89 | 89 | 89 | 89 |

- Insert 9:
  - $hash_0(9) = (9 + 0)\ \%\ 10 = 9$
  - Collision at index-9.
  - Try $i = 1, f(1) = 1^2 = 1$
  - $hash_1(9) = (9 + 1)\ \%\ 10 = 0$
  - Collision at index-0.
  - Try $i = 2, f(2) = 2^2 = 4$
  - $hash_2(9) = (9 + 4)\ \%\ 10 = 3$
  - No collision at index-3. Insert there.

# Quadratic probing

- Problem:
  - We may not be sure that we will probe all the locations of the table. (i.e. there is no guarantee to find an empty space if the table is filled more than 50% of the TableSize).
  - If the hash TableSize is not prime, the problem will be much severe!

- One theorem says:
  - If the tableSize is prime and $\lambda$<0.5 , all probes will be to different locations and an item can always be inserted.

# Some considerations

- What happens when $\lambda$ is too high:
  - Dynamically expand the table as soon as $\lambda > 0.5$, which is called ‘re-hashing’.
  - Always double the TableSize to a prime number.
  - While expanding the hash table, re-insert in the new table with a new hash-function.

- Although Quadratic Probing solves the preliminary problem of primary clustering, elements that hash to the same location will probe to the same alternative cells. This is called ‘Secondary clustering’.

# Double hashing

- Second hash-function is used to solve the collision.
  - $f(i) = i \times hash_2(x)$
- We apply a second hash function to x and probe at a distance
  $$hash_2(x), 2 \times hash_2(x) \ldots and\ so\ on.$$
- The function $hash_2(x)$ should never evaluate to Zero.
  - E.g. let $hash_2(x) = x\ mod\ 9$, and try to insert 99 in the previous example.
- A function such as $hash_2(x) = R - (x\ mod\ R)$ will work well.
  - where $R\ is\ a\ prime\ number\ <\ TableSize$.
  - E.g. try $R = 7$ for the previous example. $(7 - (x\ mod\ 7))$.

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

**Figure 5.18** Hash table with double hashing, after each insertion

- $hash_i(x) = \big(x + f(i)\big)\ mod\ 10$
- $f(i) = i \times hash_2(x)$
- $f(0) = 0$
- $hash_2(x) = R\ - (x\ mod\ R)$
- $hash_2(x) = 7\ - (x\ mod\ 7)$, when R=7

| | Empty Table | After 89 |
|---|---|---|
| 0 | | |
| 1 | | |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | | |
| 6 | | |
| 7 | | |
| 8 | | |
| 9 | | 89 |

**Figure 5.18** Hash table with double hashing, after each insertion

- Insert 89:
  - $hash_i(x) = \left(x + f(i)\right) mod\ 10$
  - Initially $i = 0$ , $f(0) = 0$
  - $hash_0(89) = (89 + 0)\ \%\ 10 = 9$
  - No collision. So insert at index-9 of hash-table.

| | Empty Table | After 89 | After 18 |
|---|---|---|---|
| 0 | | | |
| 1 | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |
| 8 | | | 18 |
| 9 | | 89 | 89 |

**Figure 5.18** Hash table with double hashing, after each insertion

- Insert 18:
  - $hash_0\ (18) = (18 + 0)\ \%\ 10 = 8$
  - No collision. So insert at index-8 of hash-table.

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

**Figure 5.18** Hash table with double hashing, after each insertion

- Insert 49:
  - $hash_0(49) = (49 + 0)\ \%\ 10 = 9$
  - Collision. Index-9 is already occupied.
  - Use $f(i) = i * hash_2(x)$ as collision resolution function.

  - For first attempt, $i = 1$.
  - $hash_2(x) = 7 - (x\ mod\ 7) = 7 - (49\ mod\ 7) = 7 - 0 = 7$

  - $f(1) = 1 * hash_2(x) = 7$
  - $hash_1(89) = \big(89 + f(1)\big)\ mod\ 10 = (89 + 7)\ \%\ 10 = 6$

  - No collision at index-6. So insert at that position.

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

**Figure 5.18** Hash table with double hashing, after each insertion

- Insert 58:
  - $hash_0(58) = (58 + 0)\ \%\ 10 = 8$
  - Collision. Index-8 is already occupied.
  - Use $f(i) = i * hash_2(x)$ as collision resolution function.

  - For first attempt, $i = 1$.
  - $hash_2(x) = 7 - (x\ mod\ 7) = 7 - (58\ mod\ 7) = 7 - 2 = 5$

  - $hash_1(58) = \big(58 + f(1)\big)\ mod\ 10 = (58 + 5)\ \%\ 10 = 3$
  - No collision at index-3. So insert at that position.

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

**Figure 5.18** Hash table with double hashing, after each insertion

- Insert 69:
  - $hash_0(69) = (69 + 0)\ \%\ 10 = 9$
  - Collision. Index-9 is already occupied.
  - $hash_2(69) = 7 - (69\ mod\ 7) = 7 - 6 = 1$
  - $hash_1(69) = \big(69 + f(1)\big)\ mod\ 10 = (69 + 1)\ \%\ 10 = 0$
  - No collision at index-0. So insert at that position.

** However, if collision occurred for $i = 1$, we need to recalculate $hash_2(x)$ for $i = 2$

# Summary

- Hash tables can be used to implement the insert and find operations in O(1).
- It depends on the Load factor (), not on the number of elements ($N$).
- It is important to have a prime TableSize and a correct choice of Load-factor and Hash-function.
- For Separate chaining,  should be close to 1.
- For Open addressing,  should not exceed 0.5, unless this will slow down the entire process.
- Re-hashing can be implemented to grow (or shrink) the table.

# References

# Thank you