cis112

Hashing

BBBF

Yeditepe University

v2024-05-27

Content

- Motivation
- Hash Table
- Hash Function
- Collision Resolution
  - Separate Chaining
  - Open Addressing
    - Linear Probing
    - Quadratic Probing
    - Double Hashing
- Implementation
- References

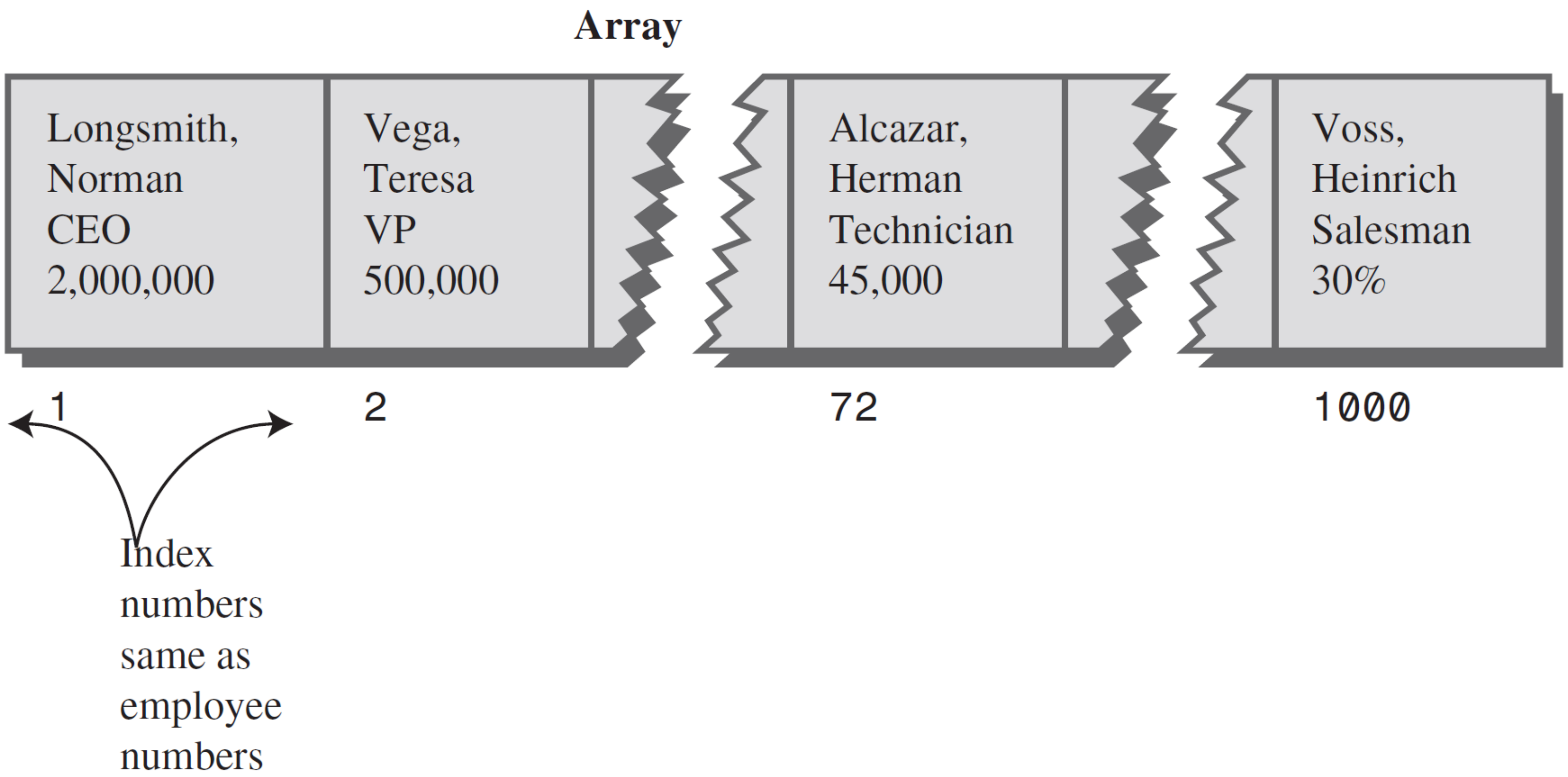
Motivation

Storing Employee Records

- Suppose you’re writing a program to access employee records for a small company with, say, 1,000 employees.
- The company’s personnel director has specified that she wants the fastest possible access to any individual record.
- Every employee has been given a number from 1 (for the founder) to 1,000 (for the most recently hired worker).
- These employee numbers can be used as keys to access the records.
- What sort of data structure should you use in this situation?

Storing Employee Records (cont.)

- One possibility is a simple array.



Dictionary

- Let's say we want to store a 50,000-word English-language dictionary in main memory.
- You would like every word to occupy its own cell in a 50,000-cell array, so you can access the word using an index number.
- What's the relationship of these index numbers to the words?

Hash Table

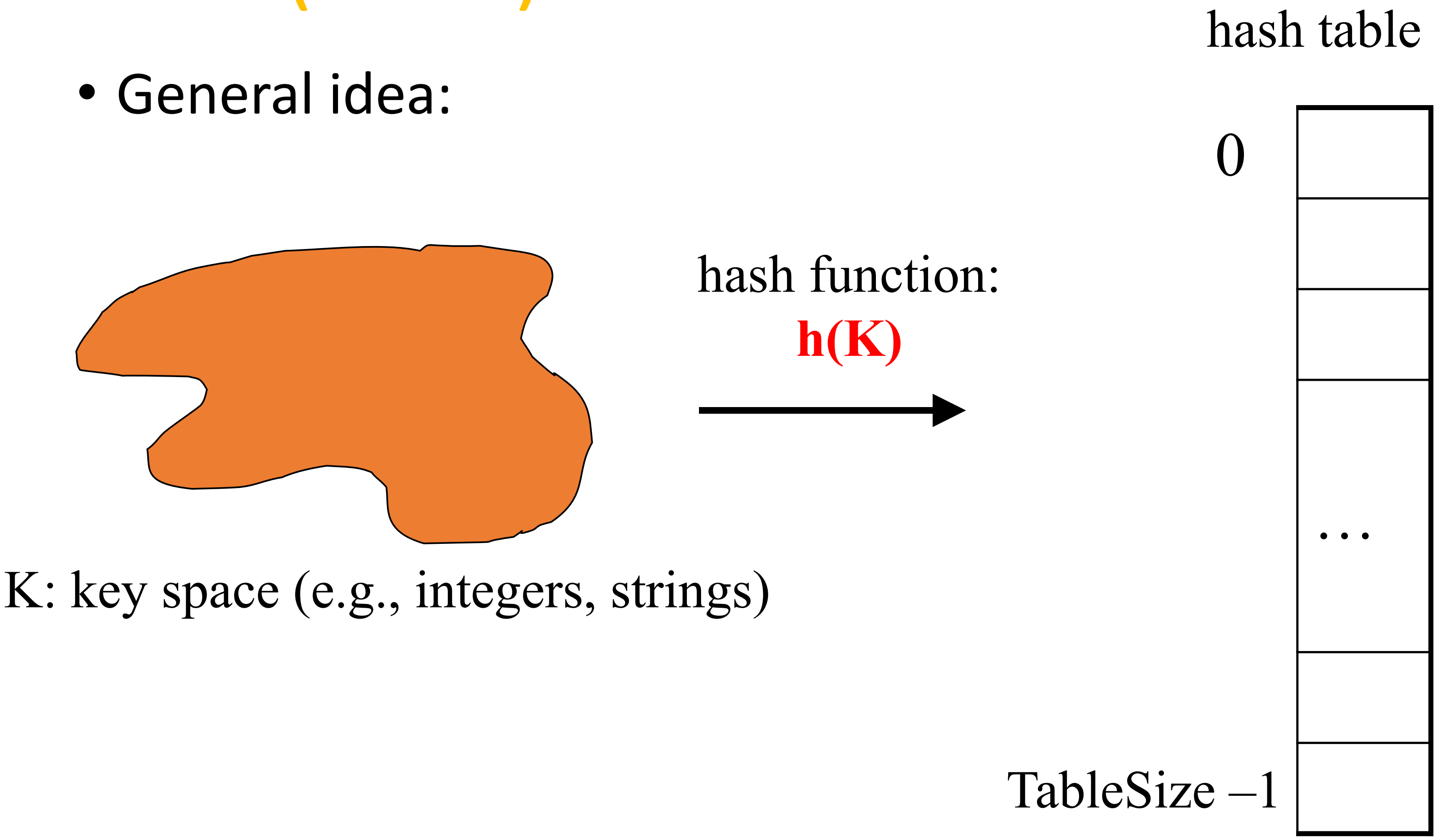
Hash Table

- A data structure that offers very fast <u>insertion</u> and <u>searching</u>.
  - can take close to constant time: O(1)
- They’re based on arrays
  - arrays are difficult to expand after they’ve been created.
- Not suitable for sorting.

- If you don’t need to visit items in order, and you can predict in advance the size of your database, hash tables are unparalleled in speed and convenience.

Hash Table (cont.)

- General idea:



Example

- key space = integers
- TableSize = 10

- **h**(K) = K mod 10

- **Insert**: 7, 18, 41, 94

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hash Table (cont.)

- Given a key *k*, we find the element whose key is *k* by just looking in the *kth* position of the array.
- This is called ***direct addressing***.
- Direct addressing is applicable when we can afford to allocate an array with one position for every possible key.
- But if we do not have enough space to allocate a location for each possible key, then we need a mechanism to handle this case.
- Another way of defining the scenario is: if we have less locations and more possible keys, then simple array implementation is not enough.

Hash Function

Hash Function

- The hash function is used to transform the key into the index. Ideally, the hash function should map
  - Each possible key to a unique slot index, but it is difficult to achieve in practice.
- Given a collection of elements, a hash function that maps each item into a unique slot is referred to as a *perfect hash function*:
  1. **simple/fast** to compute,
  2. avoid **collisions**
  3. have keys distributed **evenly** among cells.

Hash Functions

- **Truncation:**
  - e.g. 123456789 map to a table of 1000 addresses by picking 3 digits of the key.
- **Folding:**
  - e.g. 123|456|789: add them and take mod.
- **Key mod N:**
  - N is the size of the table, better if it is prime.
- **Squaring:**
  - Square the key and then truncate
- **Radix conversion:**
  - e.g. 1 2 3 4 treat it to be base 11, truncate if necessary.

Folding Example

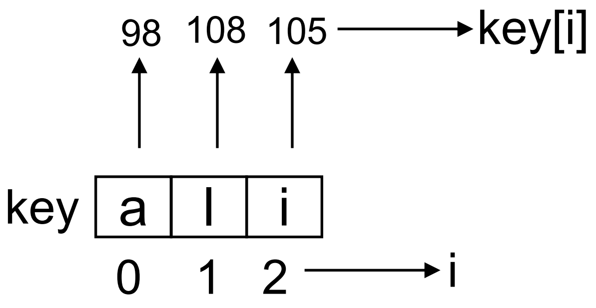
- If our element was the phone number 436-555-4601,
- we would take the digits and divide them into groups of 2 (43,65,55,46,01).
- After the addition, 43+65+55+46+01, we get 210.
- If we assume our hash table has 11 slots, then we need to perform the extra step of dividing by 11 and keeping the remainder.
- In this case 210 % 11 is 1, so the phone number 436-555-4601 hashes to slot 1.

Sample Hash Functions

- key space = strings
- $s = s_0\ s_1\ s_2 \ldots s_{k-1}$

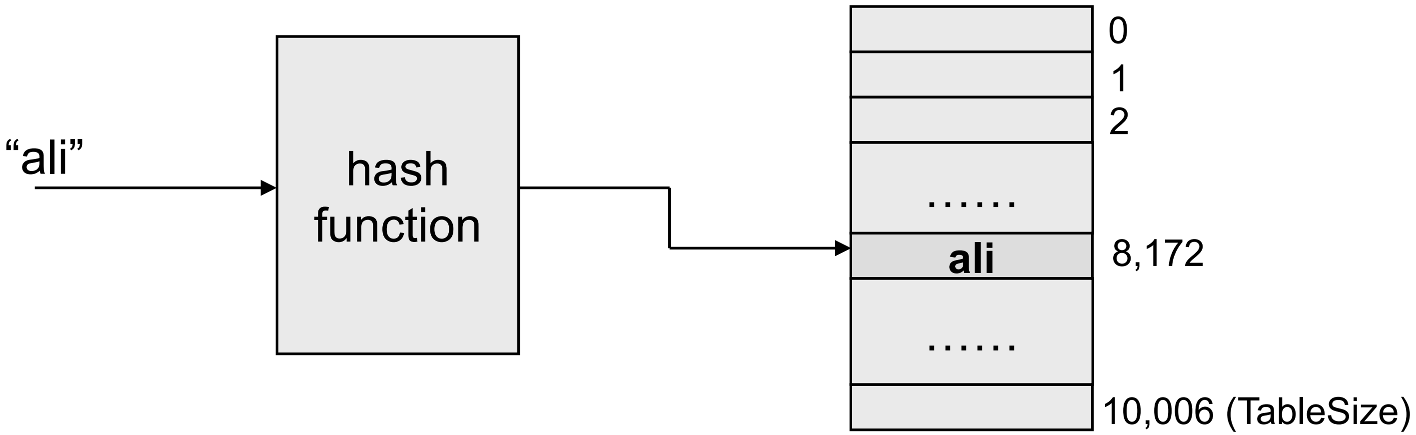
1. $h(s) = s_0 \bmod \text{TableSize}$
2. $h(s) = \left(\sum_{i=0}^{k-1} s_i \right) \bmod \text{TableSize}$
3. $h(s) = \left(\sum_{i=0}^{k-1} s_i \cdot 37^i \right) \bmod \text{TableSize}$

Hash function for strings:



KeySize = 3;

$$hash(\text{"ali"}) = (105 * 1 + 108*37 + 98*37^2) \% 10{,}007 = 8172$$



Load Factor

- The load factor of a non-empty hash table is the number of items stored in the table divided by the size of the table.
- This is the decision parameter used when we want to rehash or expand the existing hash table entries.
- This also helps us in determining the efficiency of the hashing function.
- That means, it tells whether the hash function is distributing the keys uniformly or not.

Load Factor

- Defn: The load factor, λ, of a hash table is the ratio:
- Load factor: $\lambda = \frac{N}{M}$ $\leftarrow$ N: no. of elements, M: table size
- For separate chaining,

λ = average # of elements in a bucket

Collision Resolution

Collision Resolution

Collision: when two keys map to the same location in the hash table.

Two ways to resolve collisions:

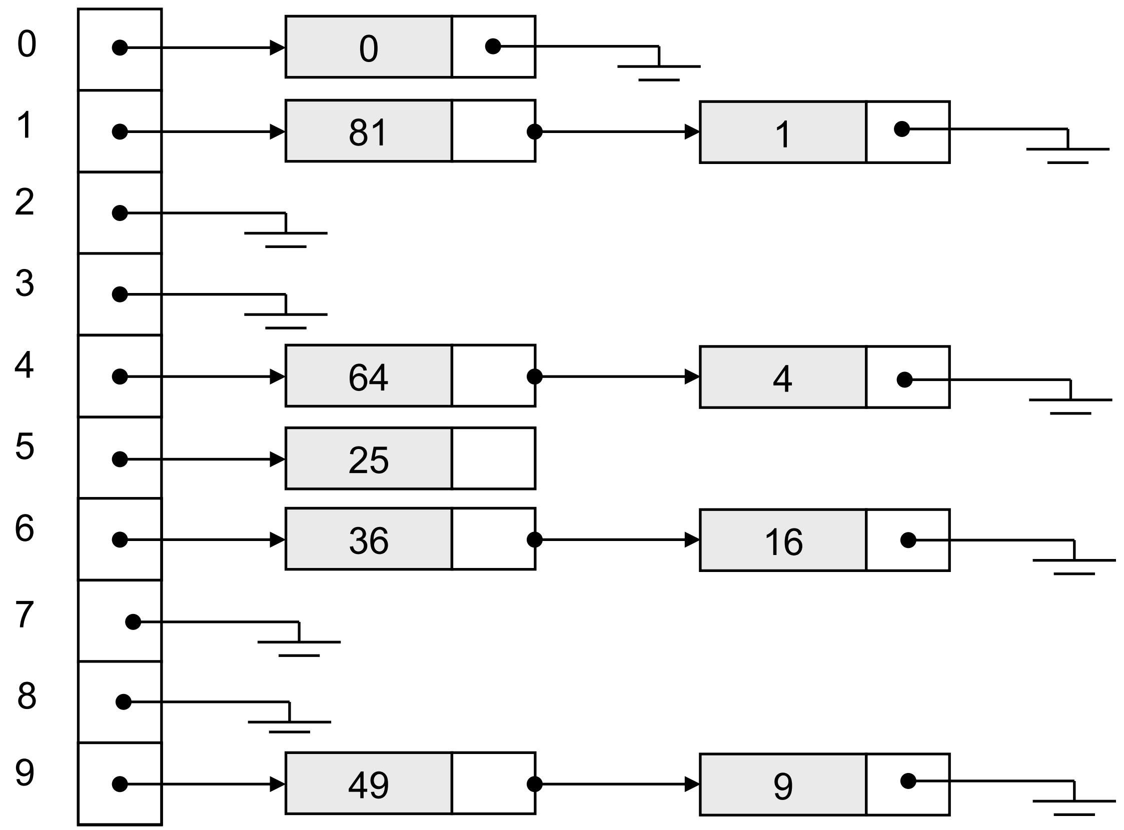
1. Separate Chaining
2. Open Addressing (linear probing, quadratic probing, double hashing)

Seperate Chaining

When two or more records hash to the same location, these records are constituted into a singly-linked list called a ***chain.***

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

hash(key) = key % 10.



tableSize: Why Prime?

- Suppose that data stored in hash table:

  7160, 493, 60, 55, 321, 900, 810

  - tableSize = 10

    data hashes to 0, 3, <u>0</u>, 5, 1, <u>0</u>, <u>0</u>

  - tableSize = 11

    data hashes to 10, 9, 5, 0, 2, <u>9</u>, 7

Operations

- **Initialization**: all entries are set to NULL
- **Find**:
  - locate the cell using hash function.
  - sequential search on the linked list in that cell.
- **Insertion**:
  - Locate the cell using hash function.
  - (If the item does not exist) insert it as the first item in the list.
- **Deletion**:
  - Locate the cell using hash function.
  - Delete the item from the linked list.

Open Addressing and Probing

Open Addressing

- In open addressing all keys are stored in the hash table itself.
- This procedure is based on probing.
  - A collision is resolved by probing.

Linear Probing

- The interval between probes is fixed at 1.
- In linear probing, we search the hash table sequentially, starting from the original hash location.
- If a location is occupied, we check the next location.
- We wrap around from the last table location to the first table location if necessary. The function for rehashing is the following:

$$\mathbf{rehash(key) = (n + 1)\ \%\ tablesize}$$

Example

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Insert:
38
19
8
109
10

- **Linear Probing**: after checking spot h(k), try spot h(k)+1, if that is full, try h(k)+2, then h(k)+3, etc.

Linear Probing

$$f(i) = i$$

- Probe sequence:

  0^{th} probe = $h(k) \bmod TableSize$

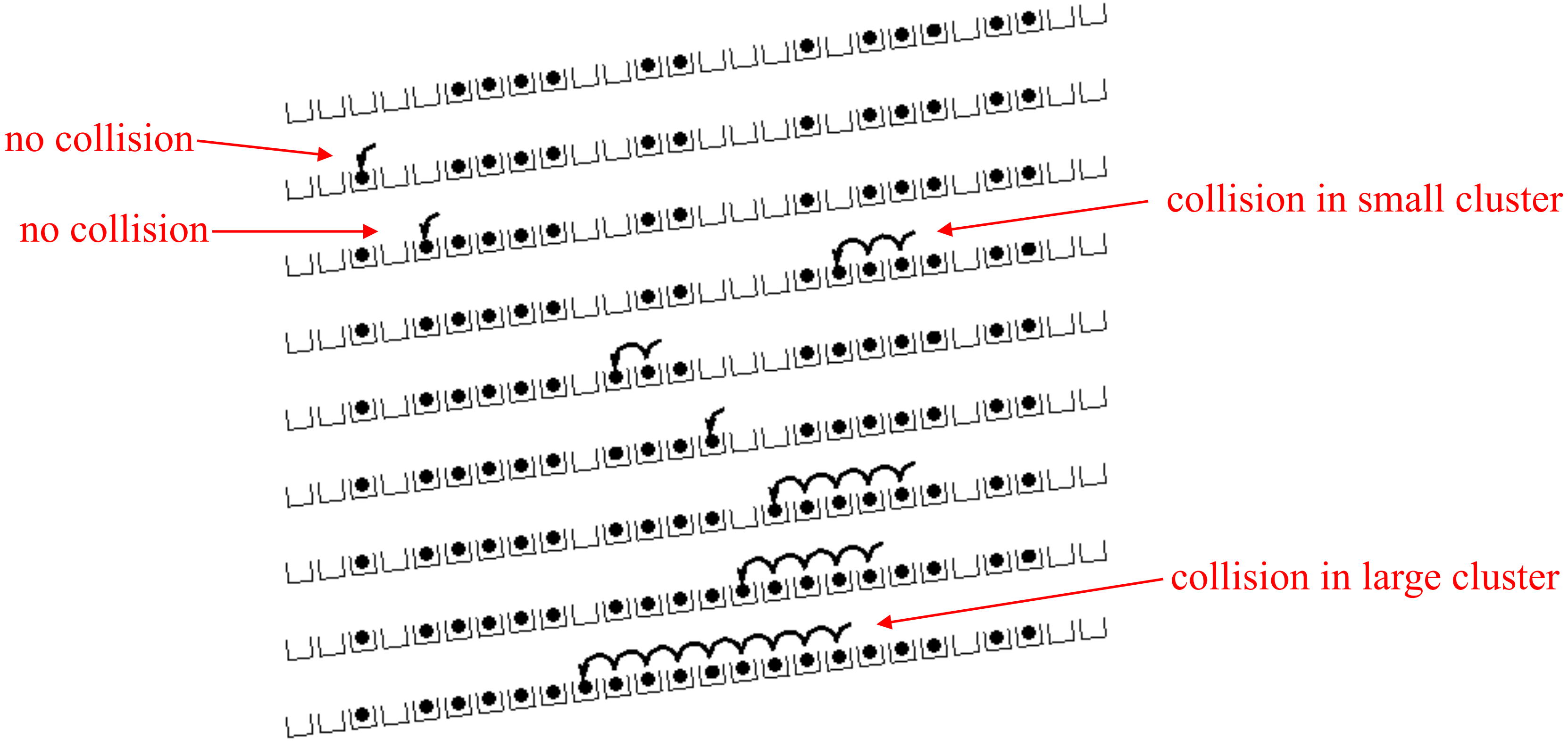
  1^{th} probe = $(h(k) + 1) \bmod TableSize$

  2^{th} probe = $(h(k) + 2) \bmod TableSize$

  . . .

  i^{th} probe = $(h(k) + i) \bmod TableSize$

Linear Probing – Clustering



[R. Sedgewick]

Load Factor in Linear Probing

- For *any* $\lambda < 1$, linear probing *will* find an empty slot
- Expected # of probes (for large table sizes)
  - successful search: $$\frac{1}{2}\left(1 + \frac{1}{(1-\lambda)}\right)$$
  - unsuccessful search: $$\frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$$
- Linear probing suffers from ***primary clustering***
- Performance quickly degrades for $\lambda > 1/2$

Quadratic Probing

Less likely to encounter Primary Clustering

$$f(i) = i^2$$

- Probe sequence:

  0^{th} probe = h(k) mod TableSize

  1^{th} probe = (h(k) + 1) mod TableSize

  2^{th} probe = (h(k) + 4) mod TableSize

  3^{th} probe = (h(k) + 9) mod TableSize

  . . .

  i^{th} probe = (h(k) + i^2) mod TableSize

Quadratic Probing

- The interval between probes increases proportionally to the hash value (the interval thus increasing linearly, and the indices are described by a quadratic function).
- The problem of clustering can be eliminated if we use the quadratic probing method.
- In quadratic probing, we start from the original hash location *i*.
- If a location is occupied, we check the locations $i + 1^2$, $i + 2^2$, $i + 3^2, i + 4^2$ …
- We wrap around from the last table location to the first table location if necessary. The function for rehashing is the following:

$$\mathbf{rehash(key)} = (\mathbf{n} + \boldsymbol{k}^2)\ \%\ \mathbf{tablesize}$$

Quadratic Probing

0	
1	
2	2
3	13
4	25
5	5
6	24
7	9
8	19
9	31
10	21

$31 \ mod \ 11 = 9$
$19 \ mod \ 11 = 8$
$2 \ mod \ 11 = 2$
$13 \ mod \ 11 = 2 \rightarrow 2 + 1^2 = 3$
$25 \ mod \ 11 = 3 \rightarrow 3 + 1^2 = 4$
$24 \ mod \ 11 = 2 \rightarrow 2 + 1^2, 2 + 2^2 = 6$
$21 \ mod \ 11 = 10$
$9 \ mod \ 11 = 9 \rightarrow 9 + 1^2, 9 + 2^2 \ mod \ 11, 9 + 3^2 \ mod \ 11 = 7$

Quadratic Probing Example

insert(76)
$76\%7 = 6$

insert(40)
$40\%7 = 5$

insert(48)
$48\%7 = 6$

insert(5)
$5\%7 = 5$

insert(55)
$55\%7 = 6$

0	
1	
2	
3	
4	
5	
6	76

But… insert(47)
$47\%7 = 5$

Quadratic Probing

- Problem:
  - We may not be sure that we will probe all locations in the table (i.e. there is no guarantee to find an empty cell if table is more than half full.)
  - If the hash table size is not prime this problem will be much severe.

- However, there is a theorem stating that:
  - If the table size is *prime* and load factor is not larger than 0.5, all probes will (guarantee) be to different locations and an item can always be inserted.

Quadratic Probing: Properties

- For *any* $\lambda < \frac{1}{2}$, quadratic probing will find an empty slot; for bigger λ, quadratic probing *may* find a slot

- Quadratic probing does not suffer from *primary* clustering: keys hashing to the same *area* are not bad

- But what about keys that hash to the same *spot*?
    - ***Secondary Clustering!***

Double Hashing

$$f(i) = i * g(k)$$

where g is a second hash function

- Probe sequence:

  0^{th} probe = h(k) mod TableSize

  1^{th} probe = (h(k) + g(k)) mod TableSize

  2^{th} probe = (h(k) + 2*g(k)) mod TableSize

  3^{th} probe = (h(k) + 3*g(k)) mod TableSize

  . . .

  i^{th} probe = (h(k) + i*g(k)) mod TableSize

Double Hashing Example

h(k) = k mod 7 and g(k) = 5 – (k mod 5)

	76	93	40	47	10	55
0						
1				47	47	47
2		93	93	93	93	93
3					10	10
4						55
5			40	40	40	40
6	76	76	76	76	76	76
Probes	1	1	1	2	1	2

Resolving Collisions with Double Hashing

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

<u>Hash Functions</u>:

$H(K) = K \bmod M$

$H_2(K) = 1 + ((K/M) \bmod (M-1))$

$M =$

Insert these values into the hash table in this order. Resolve any collisions with double hashing:

13
28
33
147
43

Rehashing

Idea: When the table gets too full, create a bigger table (usually 2x as large) and hash all the items from the original table into the new table.

- When to rehash?
  - half full ($\lambda = 0.5$)
  - when an insertion fails
  - some other threshold
- Cost of rehashing?

Implementation

Separate Chaining

```java
public class Link { // (could be other items)
    public int iData; // data item
    public Link next; // next link in list
//------------------------------------------

    public Link(int it) // constructor
    {
        iData = it;
    }
```

```java
public class HashTable {
    private SortedList[] hashArray; // array of lists
    private int arraySize;

//-------------------------------------------------
    public HashTable(int size) // constructor
    {
        arraySize = size;
        hashArray = new SortedList[arraySize]; // crea
        for (int j = 0; j < arraySize; j++) // fill ar
            hashArray[j] = new SortedList(); // with l
    }
```

```java
public void insert(Link theLink) // insert a link
{
    int key = theLink.iData;
    int hashVal = hashFunc(key); // hash the key
    hashArray[hashVal].insert(theLink); // insert at hashVal
} // end insert()
-----------------------------------------------

public void delete(int key) // delete a link
{
    int hashVal = hashFunc(key); // hash the key
    hashArray[hashVal].delete(key); // delete link
} // end delete()
-----------------------------------------------

public Link find(int key) // find link
{
    int hashVal = hashFunc(key); // hash the key
    Link theLink = hashArray[hashVal].find(key); // get link
    return theLink; // return link
}
```

Linear Probing - insert

```java
public class DataItem { // (could have more data)
    private int iData; // data item (key)
//--------------------------------------------------

    public DataItem(int ii) // constructor
    {
        iData = ii;
    }

//--------------------------------------
    public int getKey() {
        return iData;
    }
//--------------------------------------
} // end class DataItem
```

```java
public class HashTable {
    private DataItem[] hashArray; // array holds hash table
    private int arraySize;

    private DataItem nonItem; // for deleted items
//-------------------------------------------------------------------

    public HashTable(int size) // constructor
    {
        arraySize = size;
        hashArray = new DataItem[arraySize];
        nonItem = new DataItem(-1); // deleted item key is -1
    }
```

```java
  public void insert(DataItem item) // insert a DataItem
(assumes table not full)
  {
      int key = item.getKey(); // extract key
      int hashVal = hashFunc(key); // hash the key
until empty cell or -1,
      while (hashArray[hashVal] != null && hashArray[hashVal].getKey() != -1) {
          ++hashVal; // go to next cell
          hashVal %= arraySize; // wraparound if necessary
      }
      hashArray[hashVal] = item; // insert item
  } // end insert()
```

Linear Probing - delete

```java
public class DataItem { // (could have more data)
    private int iData; // data item (key)
//---------------------------------------------------

    public DataItem(int ii) // constructor
    {
        iData = ii;
    }

//---------------------------------------------------
    public int getKey() {
        return iData;
    }
//---------------------------------------------------
} // end class DataItem
```

```java
public class HashTable {
    private DataItem[] hashArray; // array holds hash table
    private int arraySize;

    private DataItem nonItem; // for deleted items
//---------------------------------------------------------------

    public HashTable(int size) // constructor
    {
        arraySize = size;
        hashArray = new DataItem[arraySize];
        nonItem = new DataItem(-1); // deleted item key is -1
    }

    public DataItem delete(int key) // delete a DataItem
    {
        int hashVal = hashFunc(key); // hash the key
        while (hashArray[hashVal] != null) // until empty cell,
        { // found the key?
            if (hashArray[hashVal].getKey() == key) {
                DataItem temp = hashArray[hashVal]; // save item
                hashArray[hashVal] = nonItem; // delete item
                return temp; // return item
            }
            ++hashVal; // go to next cell
            hashVal %= arraySize; // wraparound if necessary
        }
        return null; // can’t find item
    } // end delete()
```

Linear Probing - finding

```java
public class DataItem { // (could have more data)
    private int iData; // data item (key)
//--------------------------------------------------

    public DataItem(int ii) // constructor
    {
        iData = ii;
    }

//--------------------------------------------------
    public int getKey() {
        return iData;
    }
//--------------------------------------------------
} // end class DataItem
```

```java
public class HashTable {
    private DataItem[] hashArray; // array holds hash table
    private int arraySize;

    private DataItem nonItem; // for deleted items
//--------------------------------------------------------------

    public HashTable(int size) // constructor
    {
        arraySize = size;
        hashArray = new DataItem[arraySize];
        nonItem = new DataItem(-1); // deleted item key is -1
    }

    public DataItem find(int key) // find item with key
    {
        int hashVal = hashFunc(key); // hash the key
        while (hashArray[hashVal] != null) // until empty cell,
        { // found the key?
            if (hashArray[hashVal].getKey() == key)
                return hashArray[hashVal]; // yes, return item
            ++hashVal; // go to next cell
            hashVal %= arraySize; // wraparound if necessary
        }
        return null; // can't find item
    }
```

Double Hashing - insert

```java
public class DataItem { // (could have more data)
    private int iData; // data item (key)
//--------------------------------------------

    public DataItem(int ii) // constructor
    {
        iData = ii;
    }

//--------------------------------------------
    public int getKey() {
        return iData;
    }
//--------------------------------------------
} // end class DataItem
```

```java
public class HashTable {
    private DataItem[] hashArray; // array holds hash table
    private int arraySize;

    private DataItem nonItem; // for deleted items
//-------------------------------------------------------------

    public HashTable(int size) // constructor
    {
        arraySize = size;
        hashArray = new DataItem[arraySize];
        nonItem = new DataItem(-1); // deleted item key is -1
    }
   public int hashFunc2(int key) {
       // non-zero, less than array size, different from hF1
       // array size must be relatively prime to 5, 4, 3, and 2
       return 5 - key % 5;
   }
 public void insert(int key, DataItem item)
(assumes table not full)
 {
     int hashVal = hashFunc1(key); // hash the key
     int stepSize = hashFunc2(key); // get step size
                                     // until empty cell or -1
     while (hashArray[hashVal] != null && hashArray[hashVal].iData != -1) {
         hashVal += stepSize; // add the step
         hashVal %= arraySize; // for wraparound
     }
     hashArray[hashVal] = item; // insert item
 } // end insert()
```

Double Hashing -delete

```java
public class DataItem { // (could have more data)
    private int iData; // data item (key)
//--------------------------------------------

    public DataItem(int ii) // constructor
    {
        iData = ii;
    }

//--------------------------------------------
    public int getKey() {
        return iData;
    }
//--------------------------------------------
} // end class DataItem
```

```java
public DataItem delete(int key) // delete a DataItem
{
    int hashVal = hashFunc1(key); // hash the key
    int stepSize = hashFunc2(key); // get step size

    while (hashArray[hashVal] != null) // until empty cell,
    { // is correct hashVal?
        if (hashArray[hashVal].iData == key) {
            DataItem temp = hashArray[hashVal]; // save item
            hashArray[hashVal] = nonItem; // delete item
            return temp; // return item
        }
        hashVal += stepSize; // add the step
        hashVal %= arraySize; // for wraparound
    }
    return null; // can't find item
} // end delete()
```

```java
public class HashTable {
    private DataItem[] hashArray; // array holds hash table
    private int arraySize;

    private DataItem nonItem; // for deleted items
//------------------------------------------------------------

    public HashTable(int size) // constructor
    {
        arraySize = size;
        hashArray = new DataItem[arraySize];
        nonItem = new DataItem(-1); // deleted item key is -1
    }
   public int hashFunc2(int key) {
       // non-zero, less than array size, different from hF1
       // array size must be relatively prime to 5, 4, 3, and 2
       return 5 - key % 5;
   }
```

Double Hashing -find

```java
public class DataItem { // (could have more data)
    private int iData; // data item (key)
//--------------------------------------------

    public DataItem(int ii) // constructor
    {
        iData = ii;
    }

//--------------------------------------------
    public int getKey() {
        return iData;
    }
//--------------------------------------------
} // end class DataItem
```

```java
 public DataItem find(int key) // find item with key
(assumes table not full)
 {
     int hashVal = hashFunc1(key); // hash the key
     int stepSize = hashFunc2(key); // get step size

     while (hashArray[hashVal] != null) // until empty cell,
     { // is correct hashVal?
         if (hashArray[hashVal].iData == key)
             return hashArray[hashVal]; // yes, return item
         hashVal += stepSize; // add the step
         hashVal %= arraySize; // for wraparound
     }
     return null; // can't find item
 }
```

```java
public class HashTable {
    private DataItem[] hashArray; // array holds hash table
    private int arraySize;

    private DataItem nonItem; // for deleted items
//----------------------------------------------------------

    public HashTable(int size) // constructor
    {
        arraySize = size;
        hashArray = new DataItem[arraySize];
        nonItem = new DataItem(-1); // deleted item key is -1
    }
    public int hashFunc2(int key) {
        // non-zero, less than array size, different from hF1
        // array size must be relatively prime to 5, 4, 3, and 2
        return 5 - key % 5;
    }
```

References

- [1] R. Lafore, Data Structures & Algorithms in Java, 2nd edition, SAMS.

- [2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms. MIT Press, 2022. (CLRS)