SLIIT Academy

Higher National Diploma in Information Technology– Year 2, Semester 2

Data Structures and Algorithms

Hash Table Data Structure

Gayana Fernando

Hash Tables

Learning Objectives

LO1:	Critically analyze data structures for a given problem and use the most suitable data structure that can be use when implementing a solution for a given problem.
LO2 :	Design algorithms for solving problems that use data structure Hash Tables
LO4:	Develop the ability to compare and contrast the performance of data structures.

Introduction

- A hash table is a data structure that offers very fast insertion and searching.
- Does not depend on data items there are, insertion and searching (and sometimes deletion) can take close to constant time.

Example 1

- Program to access employee records for a small company with, say, 1,000 employees.
- Each employee record requires 1,000 bytes of storage. Thus you can store the entire database in only 1 megabyte, which will easily fit in your computer's memory.
- The company's personnel director has specified that she wants the fastest possible access to any individual record.
- Also, 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; in fact, access by other keys is deemed unnecessary.
- Employees are seldom laid off, but even when they are, their record remains in the database for reference (concerning retirement benefits and so on).
- What sort of data structure should you use in this situation?

Solution

- One possibility is a simple array. Each employee record occupies one cell of the array, and the index number of the cell is the employee number for that record.
- Accessing a specified array element is very fast if you know its index number.

Continued….

- The speed and simplicity of data access using this array-based database make it very attractive.
- However, it works in our example only because the keys are unusually well organized.
- They run sequentially from 1 to a known maximum, and this maximum is a reasonable size for an array.
- There are no deletions, so memory-wasting gaps don't develop in the sequence.
- New items can be added sequentially at the end of the array, and the array doesn't need to be very much larger than the current number of items.

Example 2

- In many situations the keys are not so well behaved as in the employee database just described.
- The classic example is a dictionary. If you want to put every word of an English language dictionary, from a to *zyzzyva* (yes, it's a word), into your computer's memory,
- Which be accessed quickly, a hash table is a good choice.

Example 3

- A similar widely used application for hash tables is in computer-language compilers, which maintain a *symbol table* in a hash table.
- The symbol table holds all the variable and function names made up by the programmer, along with the addresses where they can be found in memory.
- The program needs to access these names very quickly, so a hash table is the preferred data structure.

Hashing

- Hashing describes the process of taking some key value, and generating an array index from it.

- This is an example of a **Hash Function**. It **hashes** (converts) a number in a large range into a number in a smaller range.

ArrayIndex = hugeNumber % arraySize

- This smaller range corresponds to the index numbers in an array.
- An array into which data is inserted using a hash function is called a **Hash Table** .

Continued ….

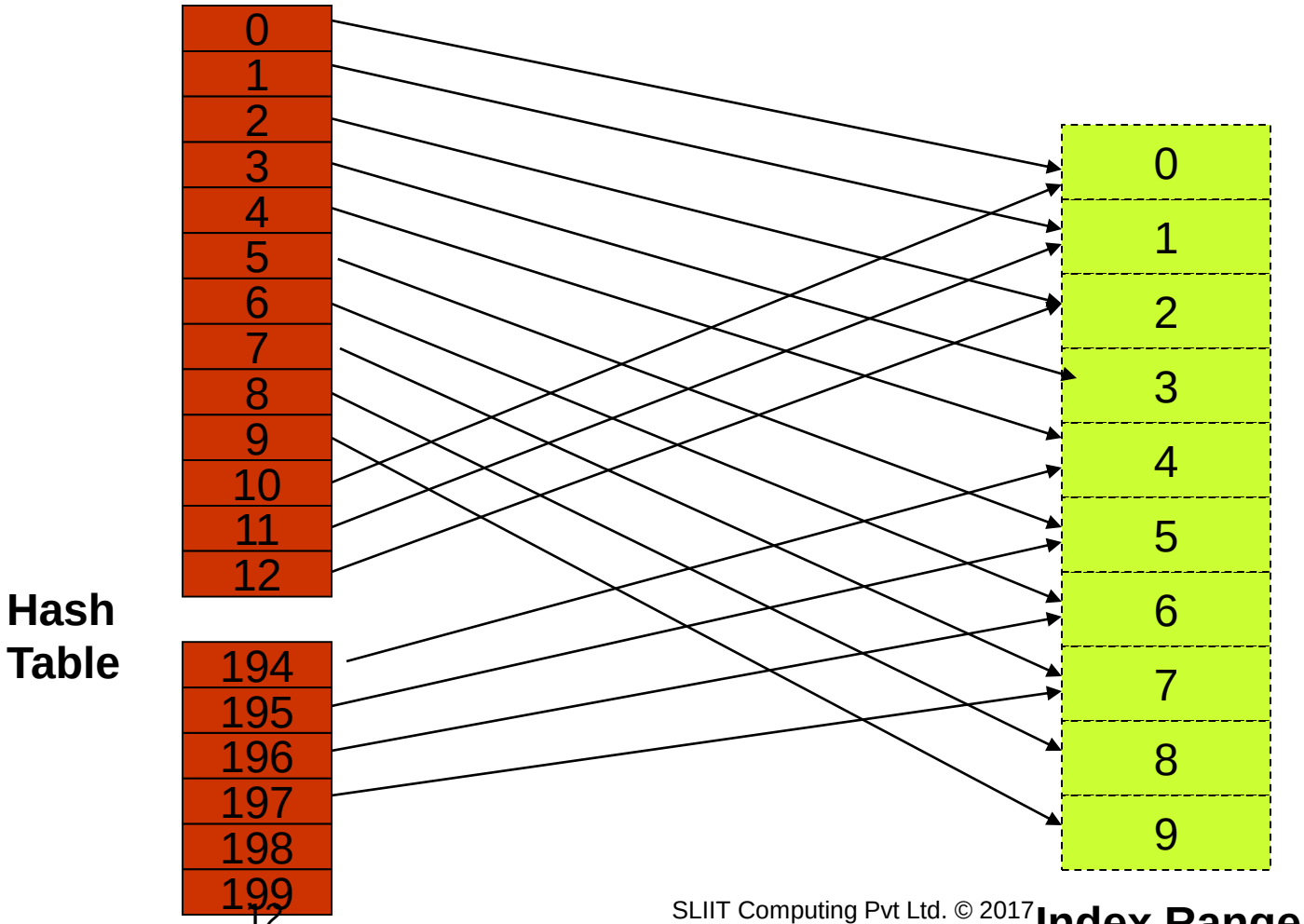
- But the trick is that for any key, there is only one hash value that is always generated.

- As an example, suppose you have to store 100 integers in an array, but index range is 0-9.
  - You want to insert a certain value, such as 134..
  - You need to generate a hash value from 0 to 9.
  - One way to do this is:

int hash = 134 % 10.

- This guarantees a value from 0 to 9, which can be our array index

Continued….



Collisions

- We pay a price for squeezing a large range into a small one.
- There's no longer a guarantee that two words won't hash to the same array index.

134 % 10 = 4
124 % 10 = 4

The hash value is the same.

- This is known as collision.

How to handle collisions

- Open addressing.
- Separate Chaining.

Open Addressing

- Have to specified an array with twice as many cells as data items.
- Thus perhaps half the cells are empty.
- One approach, when a collision occurs,
  - Search the array in some systematic way for an empty cell.
  - Insert the new item there, instead of at index specified by the hash function.
- This approach is called **open addressing**.

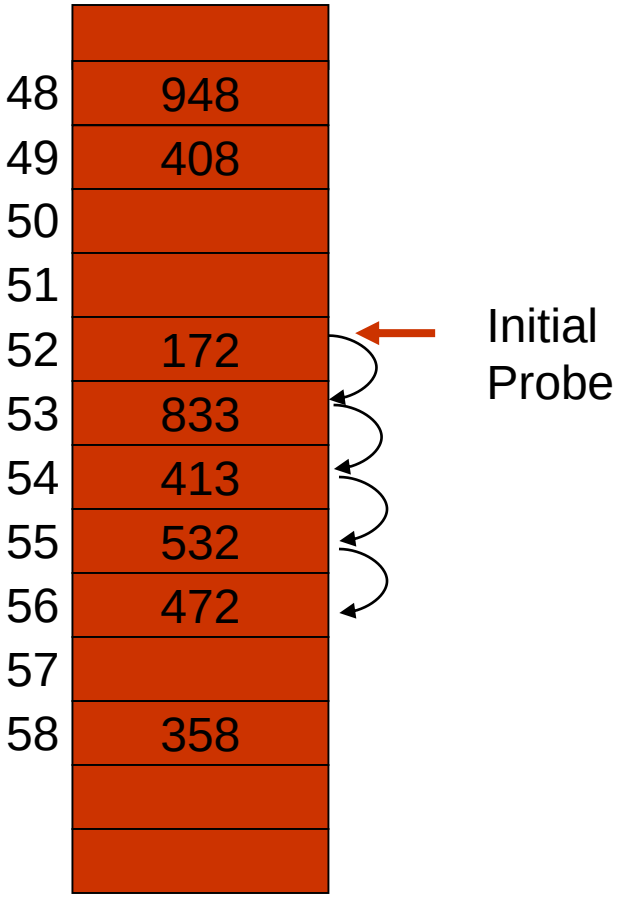
Continued

- In open addressing, when a data item can't be *placed* at the index calculated by the hash function, another location in the array is required.
- Three methods
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

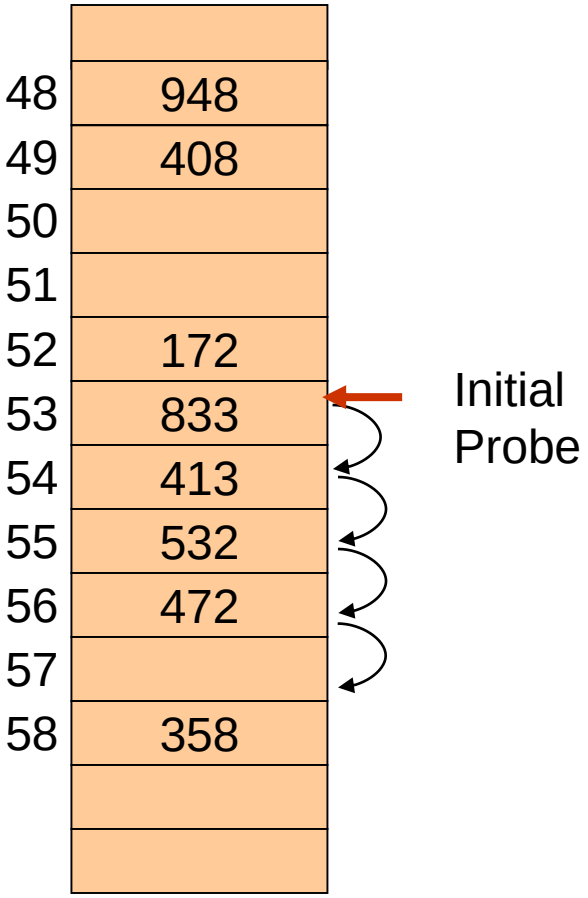
Linear Probing

- In linear probing we search sequentially for vacant cells.
- If 0 is occupied when we try to insert 1000 there, we go to 1, then 2, and so on, incrementing the index until it find an empty cell.
- This is called linear probing because it steps sequentially along line cells.

Continued…



Successful search fro 472



Unsuccessful search for 893

Clustering

- Clustering can result in very long probe lengths.
- Try inserting more items into the hash table. As it gets fuller, clusters grow larger.
- This means that it's very slow to access cells at the end of the sequence.
- The fuller the array is, the worse clustering becomes.

Clustering

- It's not a problem when the array is half full and still not too bad when it's two-thirds full.
- Beyond this, however, performance degrades seriously as the clusters grow larger and larger.
- For this reason it's critical when designing a hash table to ensure that it never becomes more than half, or at the most two-thirds, full.

Clustering

Java Code for a Linear Probing hash table

- It's not hard to create methods to handle search, insertion, and deletion with linear probe hash tables.

The hashFunc method

```
public int hashFunc (int key)
{
return key % array size;                    //hash
function
}
```

DataItem class

```
class DataItem
{
    public int iData
    public double dData;
    public String sData;

public  DataItem(int i, double d, String s)
{
    iData = i;
    dData = d;
    sData = s;
}
}
```

HashTable Class

```
class HashTable
{
     private DataItem hashArray[]; // Stores Objects
     private int arraySize;
public HashTable(int size;)
{
      hashArray = new DataItem[size];
      arraySize = size;

}
public int hashFunc (int key) {}
public DataItem find (int key){}
public void insert (DataItem item){}
public DataItem delete( int key){}
}
```

Java code for find method

```
public DataItem find (int key) / / find item with key
//(assumes table not full)
{
     int hashval = hashFunc(key); // hash the key
     while (hashArray[hashval] != null) // until empty cell,
                         // found the key?
     {
                         if (hashArray[hashval] .iData = = key)
                                   return hashArray[hashval]; // yes, return item

                         ++hashval;          // go to next cell
                          hashval %= arraySize; // wrap around if necessary
     }
return null; // can't find item
 }
```

The Find method

- The find () method first calls hashFunc () to hash the search key to obtain the index number hashval.
- The hashFunc () method applies the % operator to the search key **and** array size, as we've seen before.
- Next, in a while condition, find () checks if the item at this index is empty (null).
- If not checks if the item contains the search key. If it does, it returns the item.
- If it doesn't, find increments hashval and goes back to the top of the while loop to check if the next cell is occupied.

Java code for inset method

```
public void insert (DataItem item) //insert a DataItem (assumes table
    not full)
{
    int key = item . iData;                    // extract key hash
    int hashval = hashFunc (key)               //hash the key

while(hashArray[hashval]  != null)
    {
                ++hashval;
                Hashval %= arraySize;
    }
hashArray[hashval] = item;
}
```

The insert method

- This method uses about the same algorithm as find ( ) to locate *where* a data j should go.
- However, it's looking for an empty cell or a deleted item (null), rather than specific item.
- Once this empty cell has been located, insert ( ) places the new item into **it.**

Java code for delete method

```
public DataItem delete( int key)                    //delete a dataItem
{
      int hashval = hashFunc(key);
      while (hashArray[hashval] != null)
{
If (hashArray[hashval] . iData == key)
{
      dataItem temp = hashArray[hashval] ;
       hashArray[hashval] = null;
      return temp;
}
++hashval;
hashval %= arraySize;
}
return null
}
```

The delete method

- delete () method finds an existing item using code similar to find ( ) .
- Once the item is found delete () writes over it with null.

Quadric probing

- We've seen that clusters can occur in the linear probe approach to open addressing.
- The ratio of the number of items in a table, to the table's size, is called the *load factor.* A table with 10,000 cells and *6,667* items has a load factor of 2/3.

  load Factor = nItems / arraySize;
- Clusters can form even when the load factor isn't high. Parts of the hash table may consist of big clusters, while others are sparsely inhabited.
- Quadratic probing is an attempt to keep clusters from forming. The idea is to probe more widely separated cells, instead of those adjacent to the primary hash site.

Quadric probing

- In a linear probe, if the primary hash index is x, subsequent probes go to

  x+l, x+2, x+3, and so on.

- In quadratic probing, probes go to

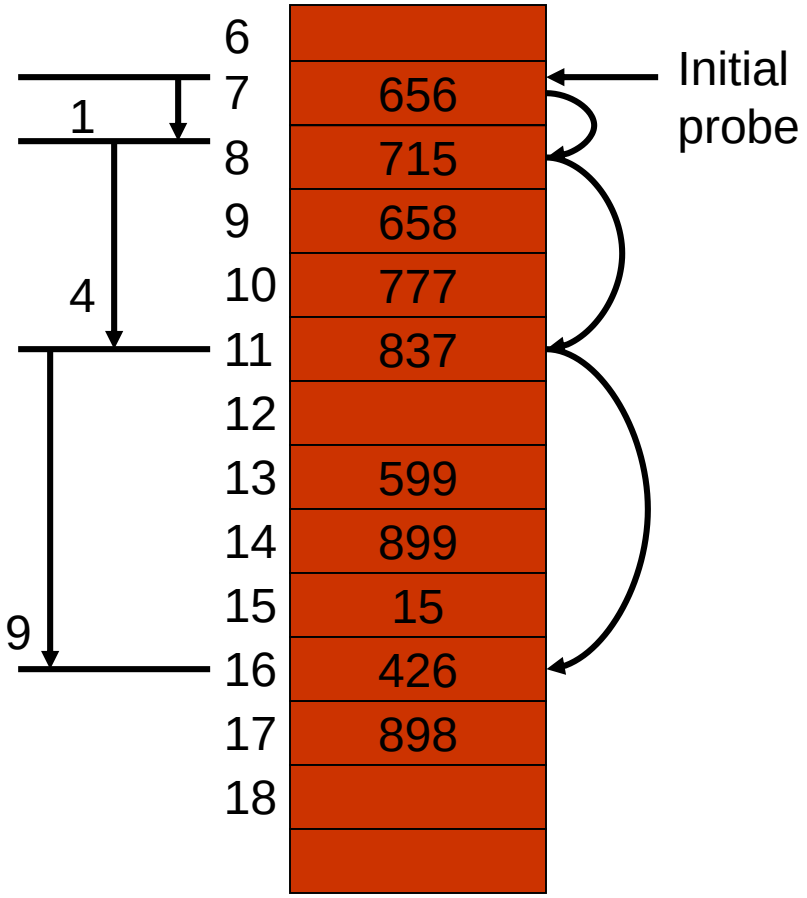
  x+1, x+4, x+9, x+16, x+25, and so on.

- The distance from the initial probe is the square of the step number:

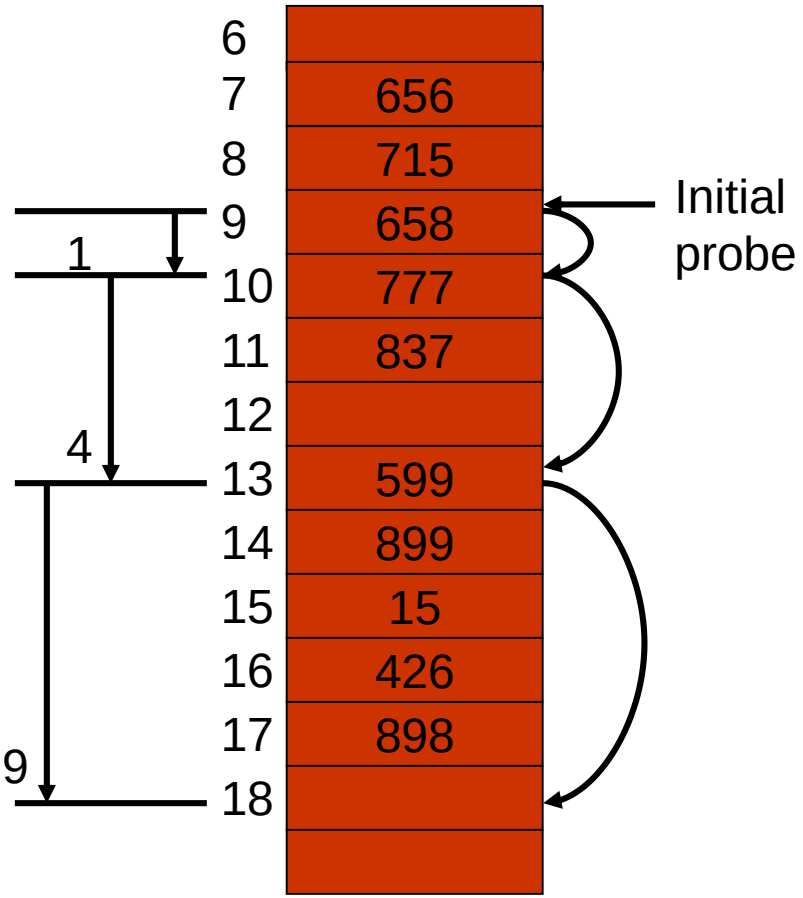
  x+12, X+22, x+32, x+42, X+52, and so on.

- Secondary clustering occurs because the algorithm that *generates the* sequence of steps in the quadratic probe always generates the same steps: 1,4,9, 16, and so on.

Continued…



Successful search for 426



Unsuccessful search for 481

Double hashing

- This approach can be used to eliminate secondary clustering as well as primary clustering.
- Its needed is a way to generate probe *sequences that depend on the key instead* being the same for every key.
- Then numbers with different keys that hash to the same index will use different probe sequences.

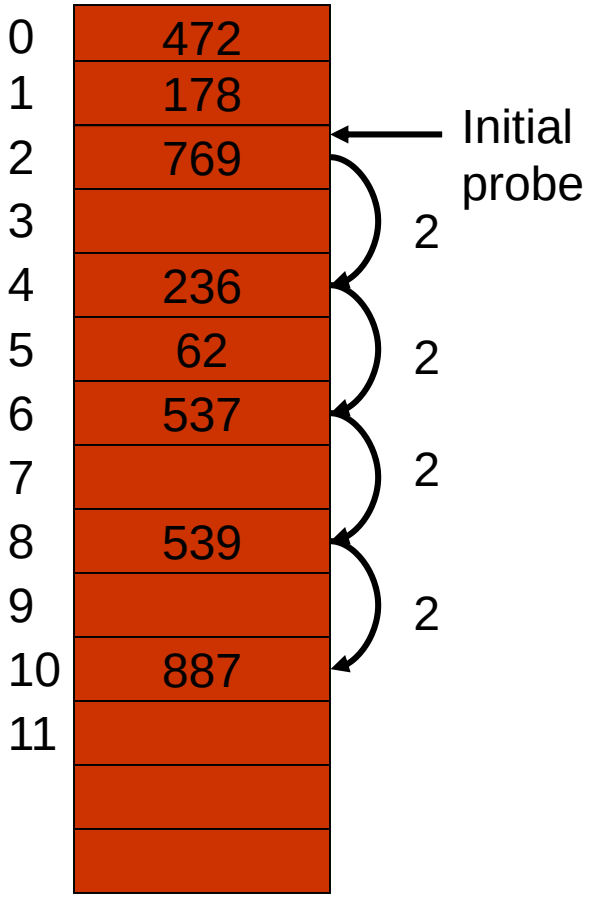
Continued…

- The solution is to hash the key a second time using a different hash function.
- Use the result as the step size.
- For a given key the step size remains constant throughout the probe, but different for different keys.
- The secondary function must
  - Not be the same as primary hashing function.
  - Never output 0.

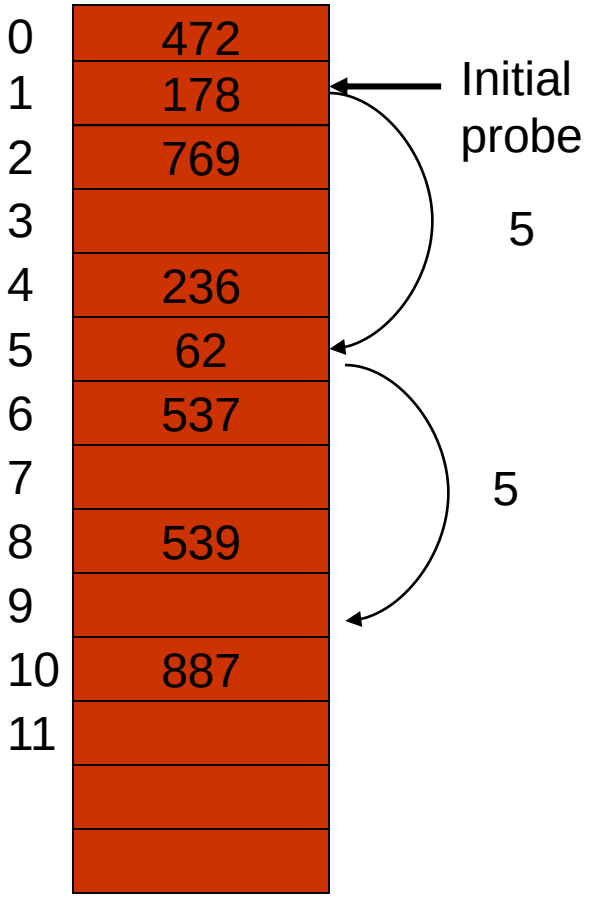
- Expert have discovered the following function will work.
  - Stepsize = constant – (key % constant)

  Where constant is a prime and smaller than the array size.
- Ex – stepSize = 5 –(key % 5)

Continued…



Successful search for 887



Successful search for 709

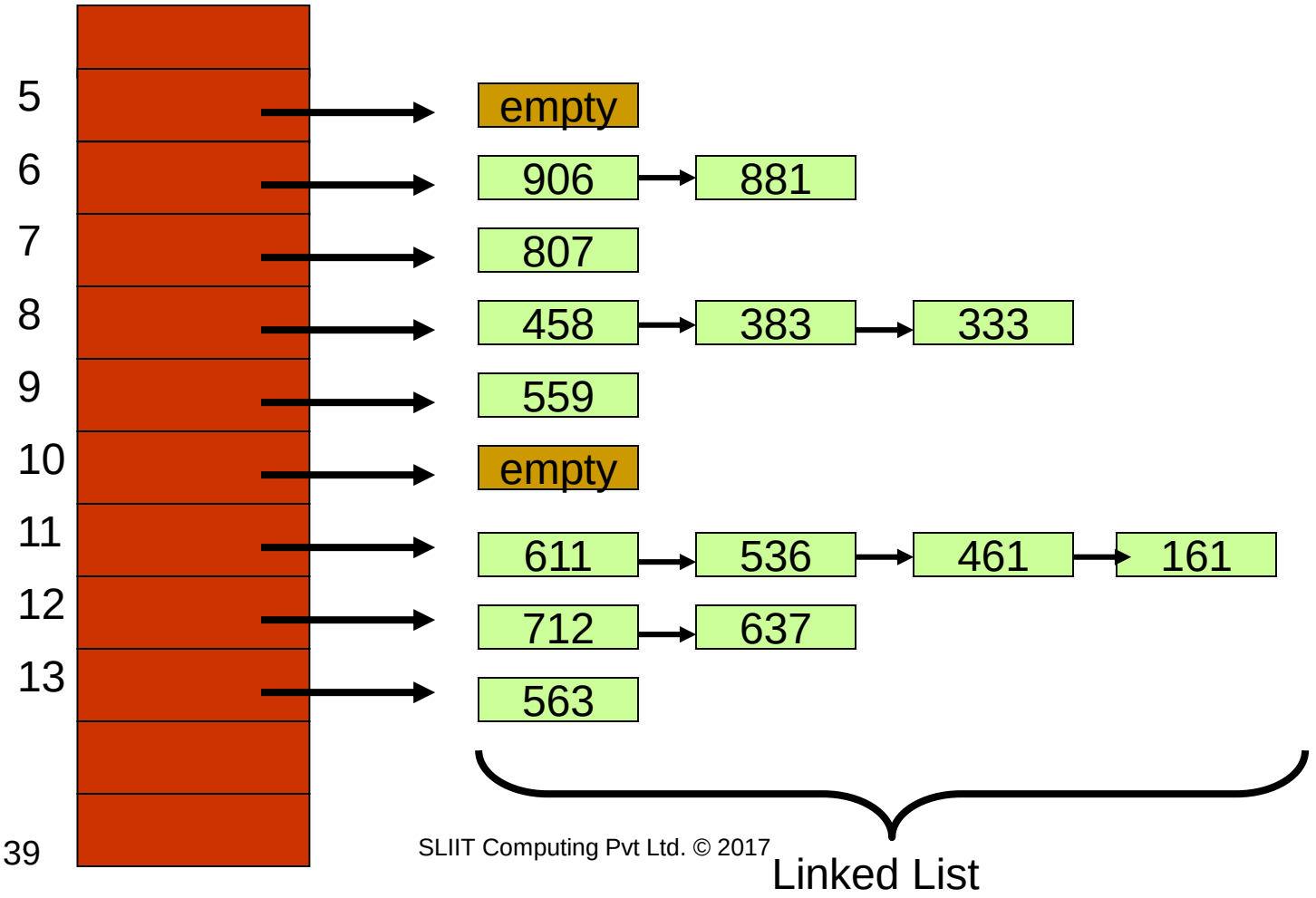
Disadvantages of Open addressing

- The loss of efficiency with high load factors is more serious for the various open addressing schemes than for separate chaining.
- In open addressing, unsuccessful searches generally take longer than successful searches.
- During a probe sequence, the algorithm can stop as soon as it finds the desired item, which is, on the average halfway through the probe sequence.
- On the other hand, it must go all the way to the end of the sequence before it's sure it can't find an item.

Separate Chaining

- In open addressing, collisions are resolved by looking for an open cell in the hash table.
- A different approach is to install a linked list at each index in the hash table.
- A data item's key is hashed to the index in the usual way, and the item is inserted into the linked list at that index. Other items that hash to the same index are simply added to the linked list; there's no need to search for empty cells in the primary array.
- Separate chaining is conceptually somewhat simpler than the various probe schemes used in open addressing.
- However, the code is longer because it must include the mechanism for the linked lists, usually in the form of an additional class

Continued…



Disadvantage of hashing

- They're based on arrays, and arrays are difficult to expand once they've been created.
- For some kinds of hash tables, performance may degrade catastrophically when the table becomes too full, so the programmer needs to have a fairly accurate idea of how many data items will need to be stored (or be prepared to periodically transfer data to a larger hash table, a time-consuming process).
- No convenient way to visit the items in a hash table in any kind of order

Hashing efficiency

- We've noted that insertion and searching in hash tables can approach 0(1) time.
- If no collision occurs, only a call to the hash function a single array reference are necessary to insert a new item or find an existing item.
- This is the minimum access time.
- If collisions occur, access times become dependent on the resulting probe lengths.
- Each cell accessed during a probe adds another time increment to the search for a vacant cell (for insertion) or for an existing cell.

Continued…

- During an access, a cell must be checked to see if it's empty, and in the case of searching or deletion if it contains the desired item.
- Thus an individual search or insertion time is proportional to the length of the probe.
- This is in addition to a constant time for the hash function.
- The average probe length (and therefore the average access time) is dependent on the load factor (the ratio of items in the table to the size of the table). As the load factor increases, probe lengths grow longer

Summary

- Hash Function
- Collision
- Hash table
  - Properties
  - Methods

Reference

- Robert Lafore - Data Structures & Algorithms in Java

Thank you!