Searching and Hashing

Hours 5
Marks 7

Syllabus

- Searching
- Introduction
- Sequential Searching
- Binary Search
- Comparison and Efficiency of Searching
- Hashing
- Probing (Linear and Quadratic)

Searching

- Searching is a process of finding an element within the list of elements stored in any order or randomly.
- Searching is divided into two categories
  - Linear Search and
  - Binary search

Sequential Search:

- In linear search, access each element of an array one by one sequentially and see whether it is desired element or not.
- A search will be unsuccessful if all the elements are accessed and the desired element is not found.
- In brief, Simply search for the given element left to right and return the index of the element, if found. Otherwise return “Not Found”.

- Analysis:

  Time complexity = O(n)

Algorithm:

```
LinearSearch(A, n, key)
{
        for(i=0; i<n; i++)
        {
                if(A[i] == key)
                return i;
        }
        return -1;                      //-1 indicates unsuccessful search
}
```

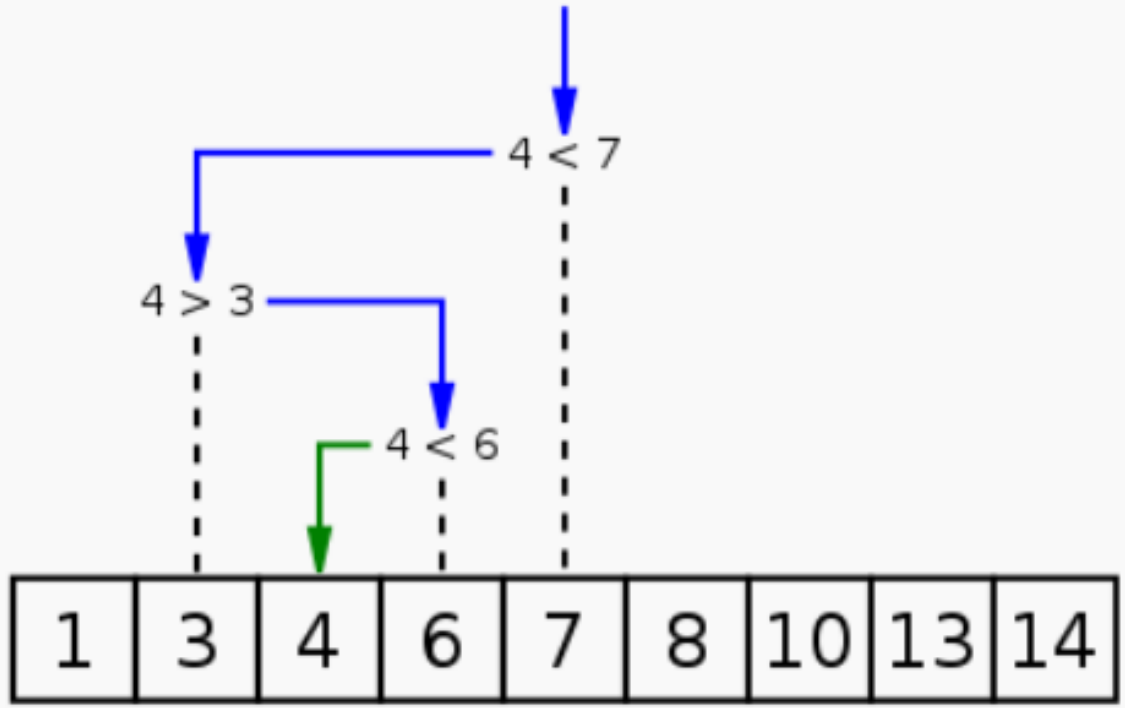
Binary Search

- Binary search is an extremely efficient algorithm.
- This search technique searches the given item in minimum possible comparisons.
- To do binary search, the array elements must be already sorted.
- Then we pick the middle element and compare with the key, if key matches then search completes otherwise we search left subarray from the key if the key is smaller than the middle element otherwise we search right subarray.

Basic Idea

- First find the middle element of the array
  - ✔ compare the middle element with an item.
  - ✔ There are three cases:
    - 1 If it is a desired element then search is successful.
    - 2 If it is less than desired item then search only the first half of the array.
    - 3 If it is greater than the desired element, search in the second half of the array.
- Repeat the same process until element is found or exhausts in the search area.
- In this algorithm every time we are reducing the search area.

Binary search algorithm



Visualization of the binary search algorithm where 4 is the target value.

Class	Search algorithm
Data structure	Array
Worst case performance	$O(\log n)$
Best case performance	$O(1)$
Average case performance	$O(\log n)$
Worst case space complexity	$O(1)$

Binary Search example

- Suppose we have an array A of 7 elements,

9	12	24	30	36	45	70
0	1	2	3	4	5	6

- Step 1:
  - Given array is in ascending order : item to be searched for is 45.
  - Beg = 0 last = 6
  - Mid = int ((beg+last)/2)= int(0+6/2) = int (3) = 3
  - So we compare A[mid] = A[3] = 30 with 45, since 45 is larger than 30, we search again in the right side of 30.
- Step 2
  - Beg 4 last 6 , Mid = (6+4)/2 = 5
  - Then we compare A[mid] = A[5] = 45 with our key which is 45.

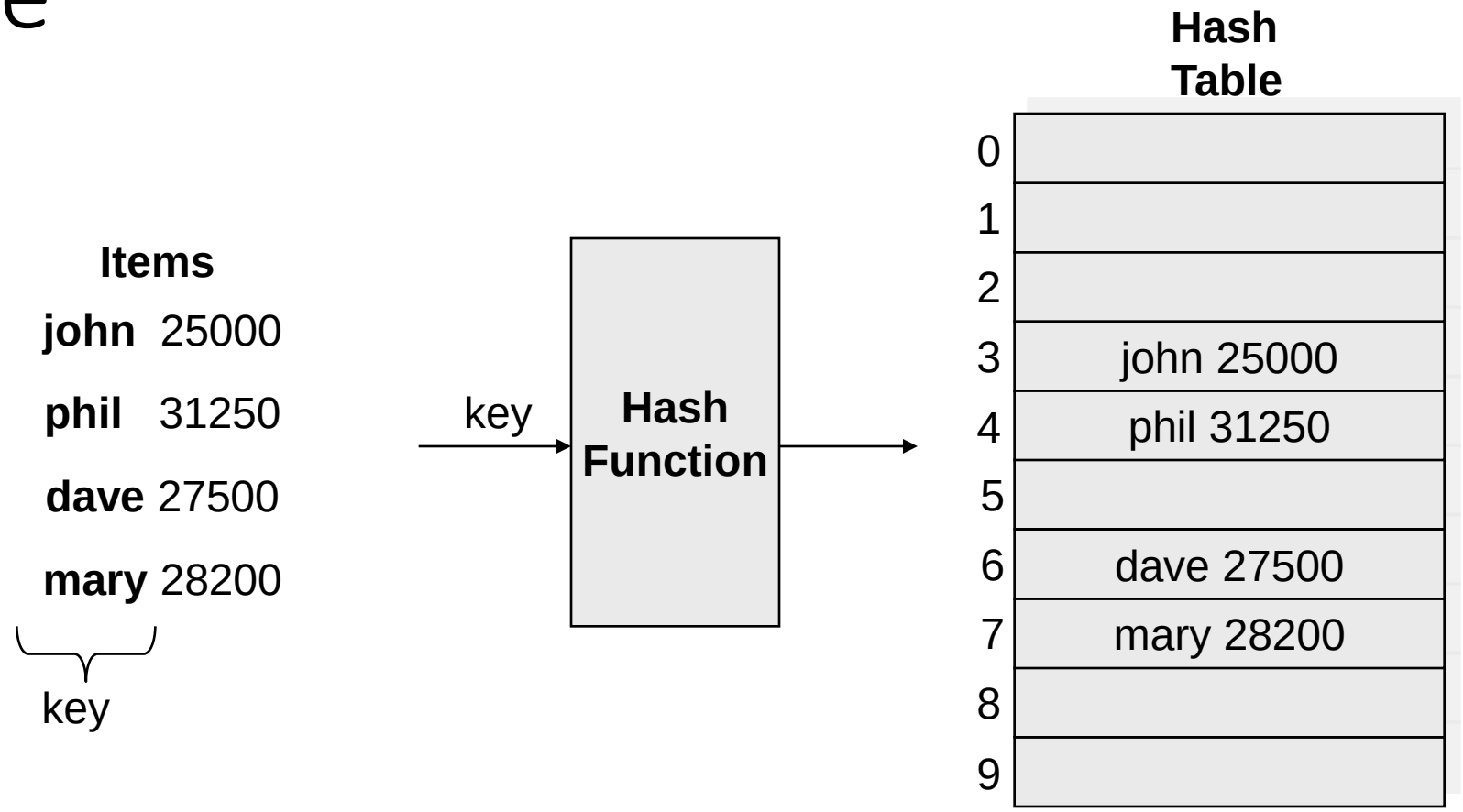
Hash Tables

- The implementation of hash tables is called **hashing**.
- Hashing is a technique used for performing **insertions**, **deletions and finds** in constant average time (i.e. O(1))
- This data structure, however, is **not efficient** in operations that require any ordering information among the elements, such as findMin, findMax and printing the entire table in sorted order.

General Idea

- The ideal hash table structure is merely an array of some fixed size, containing the items.
- A stored item needs to have a data member, called ***key***, that will be used in computing the index value for the item.
  - Key could be an *integer*, a *string*, etc
  - e.g. a name or Id that is a part of a large employee structure
- The size of the array is *TableSize*.
- The items that are stored in the hash table are indexed by values from *0* to *TableSize – 1*.
- Each key is mapped into some number in the range 0 to *TableSize – 1.*
- The mapping is called a *hash function*.

Example


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

Hash Function

- The hash function:
  - must be simple to compute.
  - must distribute the keys evenly among the cells.
- If we know which keys will occur in advance we can write *perfect* hash functions, but we don’t.

Hash function

Problems:

- Keys may not be numeric.
- Number of possible keys is much larger than the space available in table.
- Different keys may map into same location
  - Hash function is not one-to-one => collision.
  - If there are too many collisions, the performance of the hash table will suffer dramatically.

Hash Functions

- If the input keys are integers then simply *Key* mod *TableSize* is a general strategy.
  - Unless key happens to have some undesirable properties. (e.g. all keys end in 0 and we use mod 10)
- If the keys are strings, hash function needs more care.
  - First convert it into a numeric value.

Collision Resolution

- If, when an element is inserted, it hashes to the same value as an already inserted element, then we have a collision and need to resolve it.
- There are several methods for dealing with this:
  - **Separate chaining**
  - **Open addressing**
    - Linear Probing
    - Quadratic Probing
    - Double Hashing

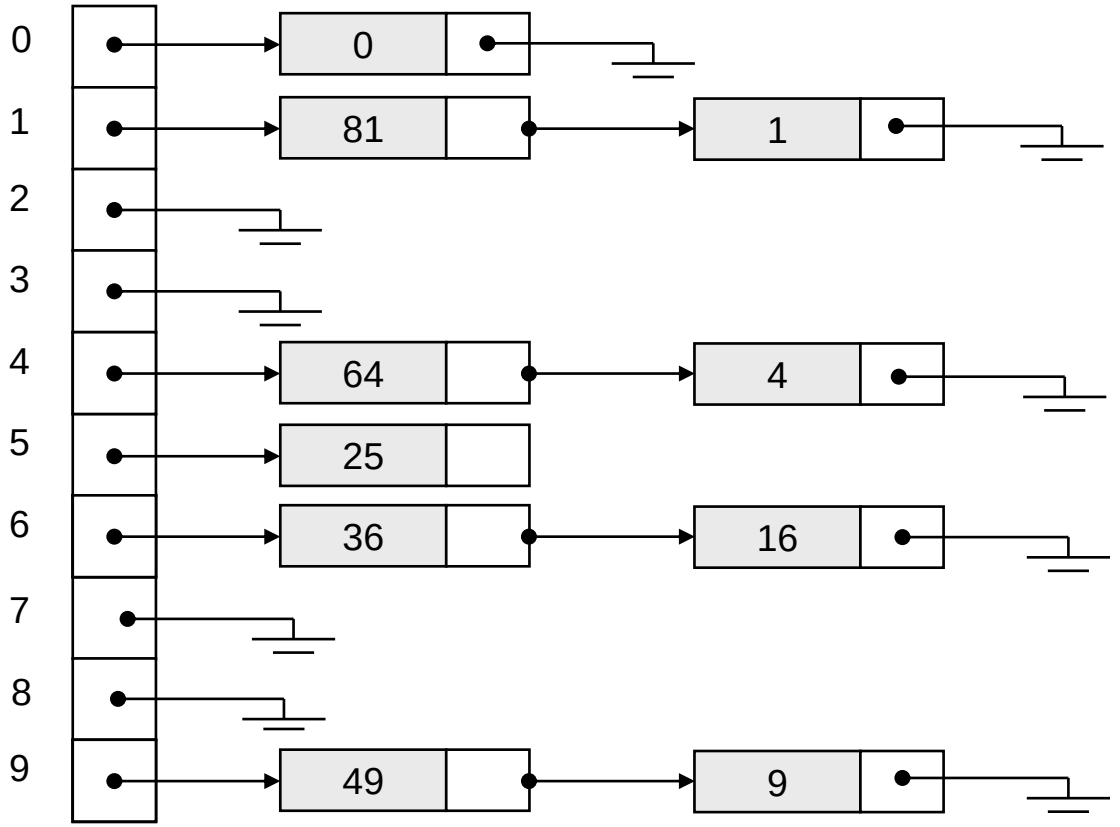
Separate Chaining

- The idea is to keep a list of all elements that hash to the same value.
  - The array elements are pointers to the first nodes of the lists.
  - A new item is inserted to the front of the list.
- Advantages:
  - Better space utilization for large items.
  - Simple collision handling: searching linked list.
  - Overflow: we can store more items than the hash table size.
  - Deletion is quick and easy: deletion from the linked list.

Example

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

hash(key) = key % 10.



Open Addressing

- Separate chaining has the disadvantage of using linked lists.
  - Requires the implementation of a second data structure.
- In an open addressing hashing system, all the data go inside the table.
  - Thus, a bigger table is needed.
  - If a collision occurs, 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 $h_i(x) = (hash(x) + f(i))$ mod *TableSize,* with $f(0) = 0$.
  - The function *f* is the collision resolution strategy.
- There are three common collision resolution strategies:
  - **Linear Probing**
  - **Quadratic probing**
  - **Double hashing**

Linear Probing

- In linear probing, collisions are resolved by sequentially scanning an array (with wraparound) until an empty cell is found.
  - i.e. *f* is a linear function of *i*, typically *f(i)= i*.
- It starts with a location where the collision occurred and does a sequential search through a hash table for the desired empty location.
- Example:
  - Insert items with keys: 89, 18, 49, 58, 9 into an empty hash table.
  - Table size is 10.
  - Hash function is hash(x) = x mod 10.
    - *f(i) = i*;

Linear probing hash table after each insertion

```
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				58	58
2					9
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Clustering Problem

- As long as table is big enough, a free cell can always be found, but the time to do so can get quite large.
- Worse, even if the table is relatively empty, blocks of occupied cells start forming.
- This effect is known as *primary clustering*.
- Any key that hashes into the cluster will require several attempts to resolve the collision, and then it will add to the cluster.

Quadratic Probing

- Quadratic Probing eliminates primary clustering problem of linear probing.
- Collision function is quadratic.
  - The popular choice is $f(i) = i^2$.
- If the hash function evaluates to h and a search in cell h is inconclusive, we try cells $h + 1^2$, $h+2^2$, … $h + i^2$.
  - i.e. It examines cells 1,4,9 and so on away from the original probe.
- Remember that subsequent probe points are a quadratic number of positions from the *original probe point*.

A quadratic probing hash table after each insertion (note that the table size was poorly chosen because it is not a prime number).

```
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

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 quadratic probing is used, and the table size is prime, then a new element can always be inserted if the table is at least half empty.

Perfect hashing

- A hash function that is injective—that is, maps each valid input to a different hash value—is said to be **perfect**. With such a function one can directly locate the desired entry in a hash table, without any additional searching.

Double Hashing

A Good Double Hash Function…

…is quick to evaluate.

…differs from the original hash function.

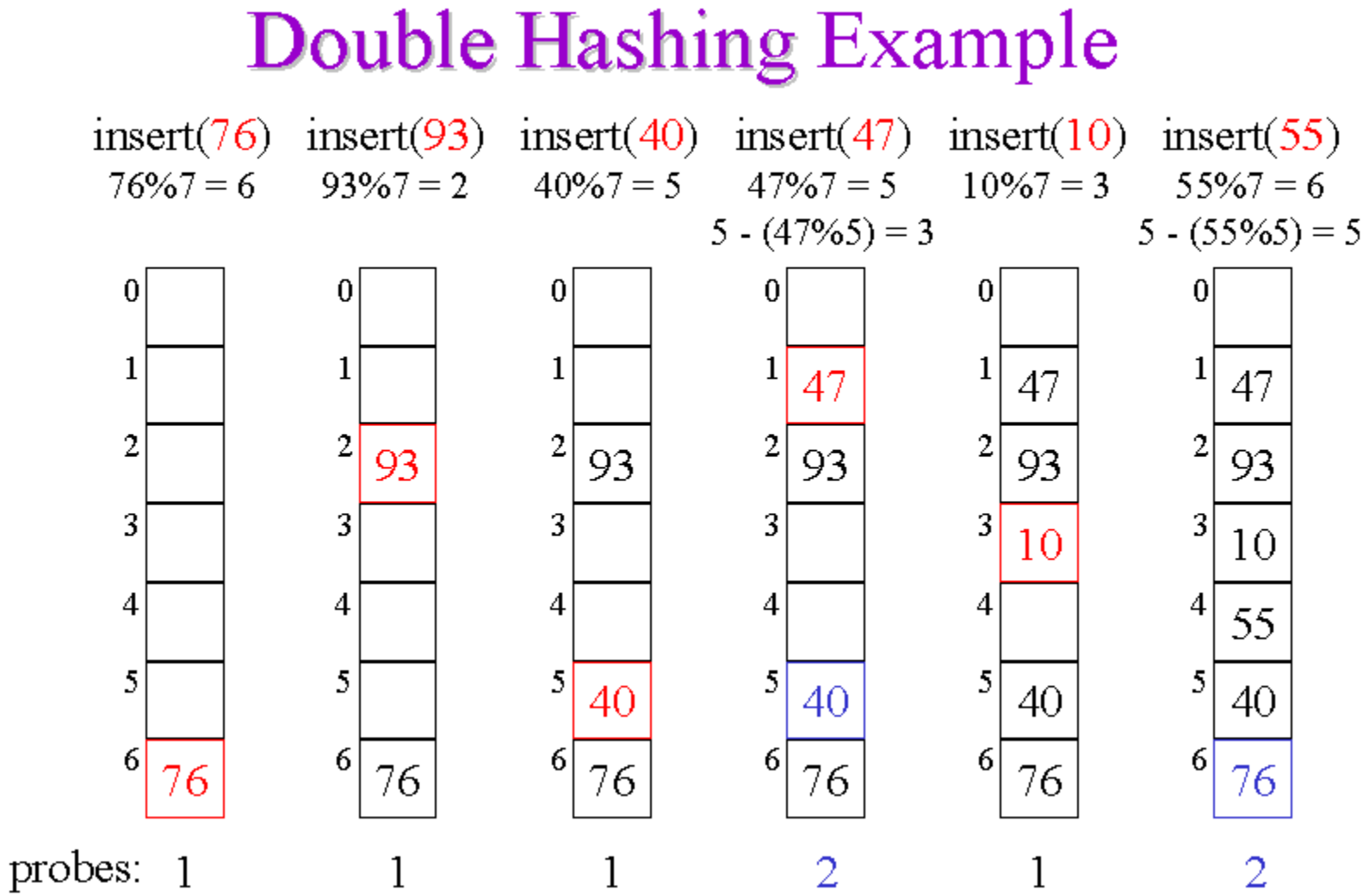
…never evaluates to 0 (mod size).

One good choice is to choose a prime R < size and:

$$hash_2(x) = R - (x \bmod R)$$

R is prime number less than table size

Double Hashing



Move 5 position from original location