COMPUTER SCIENCE DEPARTMENT FACULTY
OF ENGINEERING AND TECHNOLOGY

COMP2321

Data Structures

Chapter 5 Hashing

Introduction

- Many applications deal with lots of data Search engines and web pages
- Typical data structures like arrays, lists, and trees may not be sufficient to handle efficient lookups
- In general: When look-ups need to occur in near constant time i.e. O(1).

Why hashing?

- If data collection is sorted array, we can search for an item in O(log n) time using the binary search algorithm.
- However with a sorted array, inserting and deleting items are done in O(n) time.
- If data collection is balanced binary search tree, then inserting, searching and deleting are done in O(log n) time.
- Is there a data structure where inserting, deleting and searching for items are more efficient?
- The answer is “Yes”,
- Solution: **Hashing**
- In fact hashing is used in: Web searches, Spell checkers Databases, Compilers, passwords, etc.

Def. Of Hashing

- Hashing is a technique used for performing insertions, deletions and finds in constant average time (i.e. O(1))
- A hash function is a function that can be used to map data of arbitrary size onto data of a fixed size.
- 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.

Basic definitions

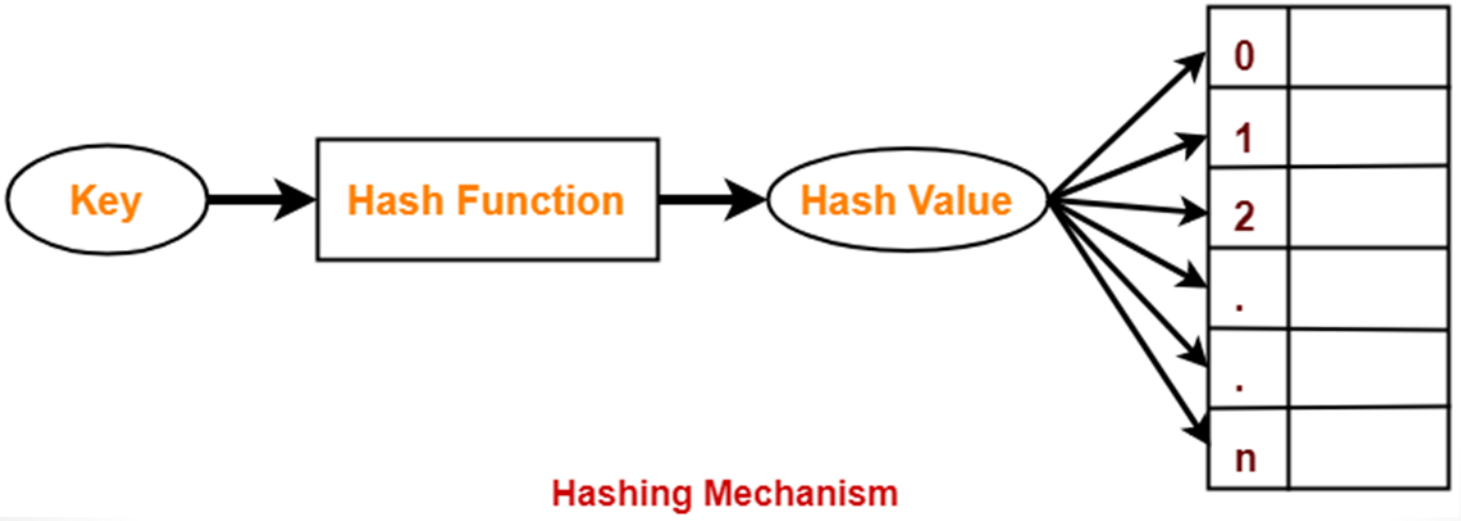
- Problem definition:

  Given a set of items (**S**) and a given item (**i**), define a data structure that supports operations such as find/insert/delete **i** in constant time.

- A solution:

  A hashing function **h** maps a large data set into a small index set.

  Typically the function involves the **mod( )** operation.



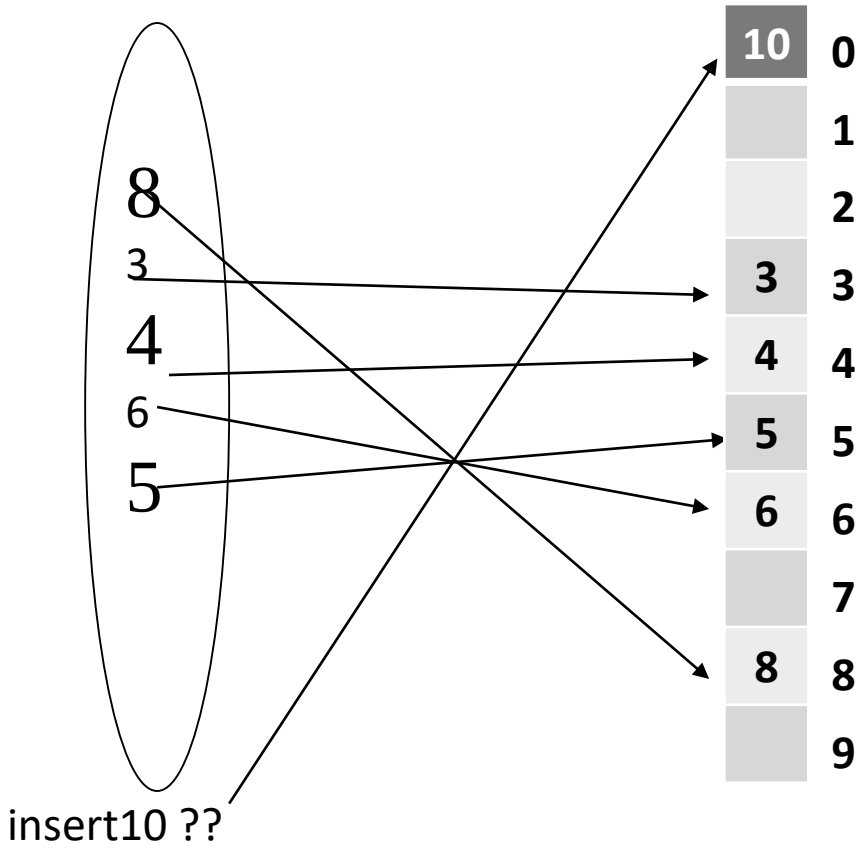
Hashing Mechanism

Design of a hash function

- Two concerns:
  a. The hash function should be simple enough.
  b. The hash function should **distribute** the data items evenly over the whole array.
- Why?
  - For (a): efficiency
  - For (b): to avoid **collision**, and to make good use of array space.

Simple hash function

Hash Function h(X) = (X % TableSize)



insert 13 ?? → **Collision**

Collision Resolution Techniques

BIRZEIT UNIVERSITY

- There are two broad ways of collision resolution:

1. Separate Chaining: An array of linked list implementation.

2. Open Addressing: Array-based implementation.

   (i) Linear probing (linear search)
   (ii) Quadratic probing (nonlinear search)
   (iii) Double hashing (uses two hash functions)

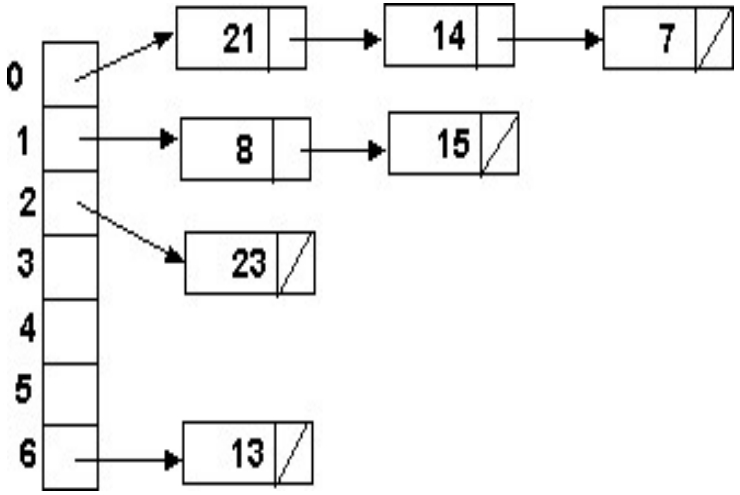
Separate Chaining (Closed Addressing)

- The hash table is implemented as an array of linked lists.
- Inserting an item, `r`, that hashes at index i is simply insertion into the linked list at position `i`.
- Synonyms are chained in the same linked list.
- Retrieval of an item, `r`, with hash address, `i`, is simply retrieval from the linked list at position `i`.
- Deletion of an item, `r`, with hash address, `i`, is simply deleting `r` from the linked list at position `i`.

Separate Chaining (Closed Addressing)

- **Example:** Load the keys **23, 14, 13, 21, 8, 7, and 15** , in this order, in a hash table of size **7** using separate chaining with the hash function:

h(key) =key % Table_Size



Time complexity > O(1)
but less than log(N)

Separate Chaining (key is string)

- Use the hash function **hash** to load the following items into a hash table of size **13** using separate chaining:

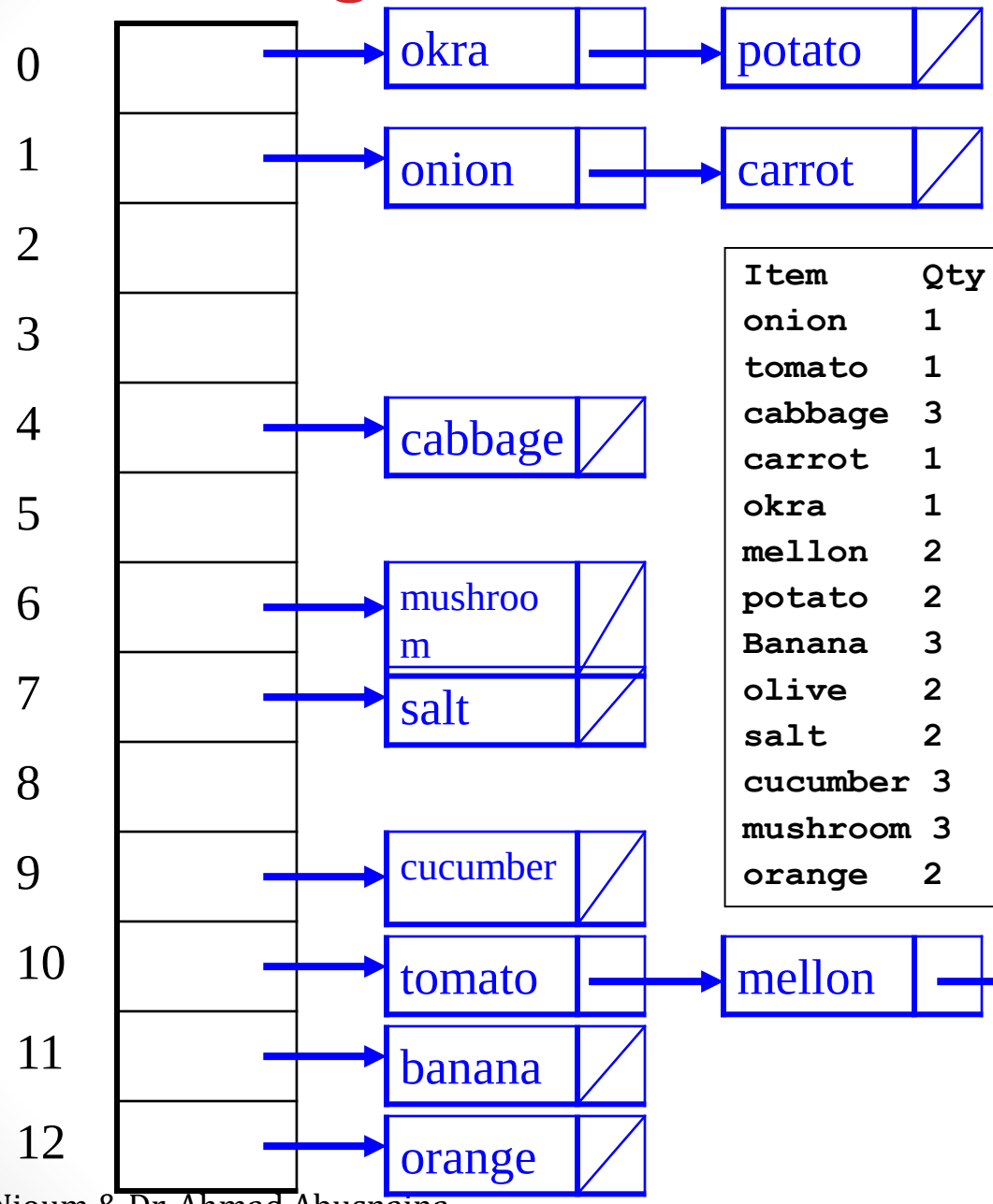
onion	1	10.0
tomato	1	8.50
cabbage	3	3.50
carrot	1	5.50
okra	1	6.50
mellon	2	10.0
potato	2	7.50
Banana	3	4.00
olive	2	15.0
salt	2	2.50
cucumber	3	4.50
mushroom	3	5.50
orange	2	3.00

- Solution:

character	a	b	c	e	g	h	i	k	l	m	n	o	p	r	s	t	u	v
ASCII code	97	98	99	101	103	104	105	107	108	109	110	111	112	114	115	116	117	118

hash(onion) = (111 + 110 + 105 + 111 + 110) % 13 = 547 % 13 = 1
hash(salt) = (115 + 97 + 108 + 116) % 13 = 436 % 13 = 7
hash(orange) = (111 + 114 + 97 + 110 + 103 + 101)%13 = 636 %13 = 12

Separate Chaining (key is string)



Item	Qty	Price	h(key)
onion	1	10.0	1
tomato	1	8.50	10
cabbage	3	3.50	4
carrot	1	5.50	1
okra	1	6.50	0
mellon	2	10.0	10
potato	2	7.50	0
Banana	3	4.0	11
olive	2	15.0	10
salt	2	2.50	7
cucumber	3	4.50	9
mushroom	3	5.50	6
orange	2	3.00	12

Open Addressing

- All items are stored in the hash table itself.
- In addition to the cell data (if any), each cell keeps one of the three states: EMPTY, OCCUPIED, DELETED.
- While inserting, if a collision occurs, alternative cells are tried until an empty cell is found.

- **Deletion**: (lazy deletion): When a key is deleted the slot is marked as **DELETED rather than EMPTY** otherwise subsequent searches that hash at the deleted cell will fail.

- **Probe sequence**: A probe sequence is the sequence of array indexes that is followed in searching for an empty cell during an insertion, or in searching for a key during find or delete operations.

Open Addressing

- **Probe sequence**: A probe sequence is the sequence of array indexes that is followed in searching for an empty cell during an insertion, or in searching for a key during find or delete operations.

- The most common probe sequences are of the form:

$$h_i(key) = [h(key) + f(i)] \% \ n, \quad \text{for } i = 0, 1, \ldots, n-1.$$

  where **h** is a hash function and **n** is the size of the hash table

- The function **f(i)** is required to have the following two properties:

  Property 1: f(0) = 0

  Property 2: The set of values **{f(0) % n, f(1) % n, f(2) % n, . . . ,f(n-1) % n}** must contain every integer between **0** and **n - 1** inclusive.

Open Addressing

- The function **f(i)** is used to resolve collisions. $h_i(key) = [h(key) + f(i)] \% n$, for $i = 0, 1, \ldots, n-1$.
- To insert item r, we examine array location $h_0(r) = h(r)$. If there is a collision, array locations $h_1(r), h_2(r), \ldots, h_{n-1}(r)$ are examined until an **empty slot is found**.
- Similarly, to find item **r**, we examine the same sequence of locations in the same order.

Note: For a given hash function **h(key)**, the only difference in the open addressing collision resolution techniques (linear probing, quadratic probing and double hashing) is in the definition of the function **f(i)**.

- **Types of Open Addressing:**

Collision resolution technique	f(i)
Linear probing	i
Quadratic probing	i^2
Double hashing	$i*h_p(key)$ where $h_p(key)$ is another hash function.

Open Addressing Facts

- In general, primes give the best table sizes.
- With any open addressing method of collision resolution,
  as the table fills, there can be a severe degradation in the table performance.
- Load factors (λ) between 0.6 and 0.7 are common.
  λ =(Number of element/Table Size)
- Load factors > 0.7 are undesirable.
- The search time depends only on the load factor, *not* on the table size.
- We can use the desired load factor to determine appropriate table size:

$$\text{table size} = \text{smallest prime} \geq \frac{\text{number of items in table}}{\text{desired load factor}}$$

Open Addressing: Linear Probing

- **F(i)** is a linear function in **i** of the form **F(i) = i**.
- Usually **F(i)** is chosen as:

  f(i) = i **for i = 0, 1, . . . , tableSize – 1**

- The probe sequences are then given by:

  h_i(key) = [h(key) + i] % tableSize for i = 0, 1, . . . , tableSize – 1

Open Addressing: Linear Probing

BIRZEIT UNIVERSITY

Example: Perform the operations given below, in the given order, on an initially empty hash table of size **13** using linear probing.

The hash function: **h(key) = key % 13**:

insert(18), insert(26), insert(35), insert(9), find(15), insert(48), delete(35), delete(40), find(9), insert(64), insert(47), find(35)

- The required probe sequences are given by:

$$h_i(key) = (h(key) + i) \% 13 \qquad i = 0, 1, 2, \ldots, 12$$

$$= (key\%13 + i) \% 13$$

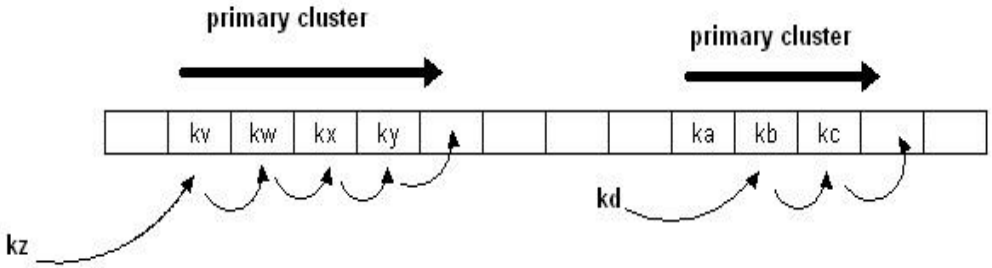
Solution At board

Operation	Probe Sequence	comment
Insert(18)	5	success
Insert(26)	0	success
Insert(35)	9	success
Insert(9)	9	Collison
	10	success
Find(15)	2	Failed /Empty status
Insert(48)	9	Collison
	10	Collison
	11	success
Delete(35)	9	Success, deleted but key not removed. Status changed to D
Find(9)	9	The search continued, location 9 doesn't contains 9
	10	success
Insert(64)	12	success
Insert(47)	8	success
Find(35)	9	Failed, location 9 is their but status changed to D

Index	Status	Value
0	O	26
1	E	
2	E	
3	E	
4	E	
5	O	18
6	E	
7	E	
8	O	47
9	D	35
10	O	9
11	O	48
12	O	64

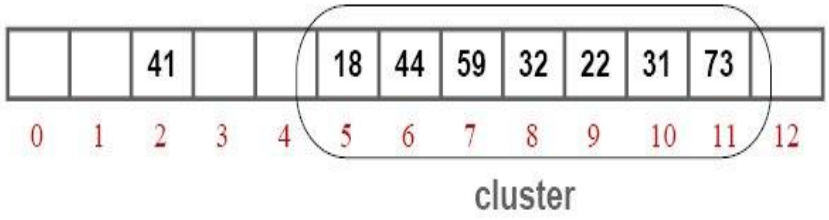
Disadvantage of Linear Probing: Primary Clustering

- Linear probing is subject to a primary clustering phenomenon.
- Elements tend to cluster around table locations that they originally hash to.
- Primary clusters can combine to form larger clusters. This leads to long probe sequences and hence deterioration in hash table efficiency.



Example of a primary cluster: Insert keys: **18, 41, 22, 44, 59, 32, 31, 73**, in this order, in an originally empty hash table of size **13**, using the hash function **h(key) = key % 13** and **f(i) = i**:

h(18) = 5
h(41) = 2
h(22) = 9
h(44) = 5+1
h(59) = 7
h(32) = 6+1+1
h(31) = 5+1+1+1+1+1
h(73) = 8+1+1+1



Open Addressing: Quadratic Probing

- Quadratic probing eliminates primary clusters.
- **f(i)** is a quadratic function in **i** of the form

  $f(i) = i^2$ **for i = 0, 1, . . . , tableSize – 1**

- The probe sequences are then given by:

  $$h_i(key) = [h(key) + i^2] \% tableSize$$ **, for i = 0, 1, . . . , tableSize – 1**

Open Addressing: Quadratic Probing

BIRZEIT UNIVERSITY

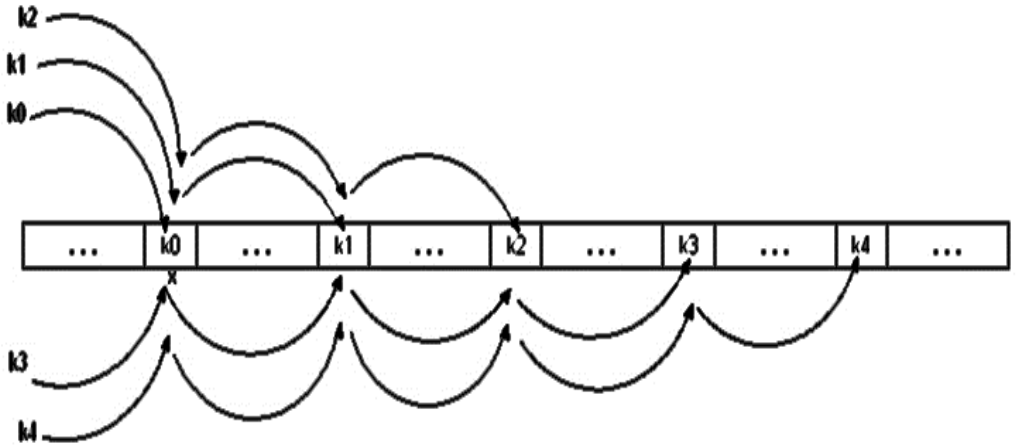
- Example: Load the keys **23, 13, 21, 14, 7, 815**,34 **, and 47** in this order, in a hash table of size **13** using quadratic probing with
- **f(i) = i^2** and the hash function: **h(key) = key % 13**
- The required probe sequences are given by:

$$h_i(key) = (h(key) + i^2) \% 13 \quad i = 0, 1, 2, 3$$

Solution in class At board

Secondary Clusters

- Quadratic probing is better than linear probing because it eliminates primary clustering.
- However, it may result in **secondary clustering**: if **h(k1) = h(k2)** the probing sequences for **k1** and **k2** are exactly the same.
  This sequence of locations is called a secondary cluster.
- Secondary clustering is less harmful than primary clustering because secondary clusters do not combine to form large clusters.
- **Example of Secondary Clustering:** Suppose keys **k0, k1, k2, k3, and k4** are inserted in the given order in an originally empty hash table using **quadratic probing** with **f(i) = i^2**. Assuming that each of the keys hashes to the same array index **x**. A secondary cluster will develop and grow in size:



Open Addressing: Double Hashing

- To eliminate secondary clustering, synonyms must have different probe sequences.
- Double hashing achieves this by having two hash functions that both depend on the hash key.
- $f(i) = i * h_p(key)$ **for i = 0, 1, . . . , tableSize – 1**
  where h_2 is another hash function.
- The probing sequence is:
  $$h_i(key) = [h(key) + i*h_p(key)]\% \text{ tableSize}$$ **for i = 0, 1, . . . , tableSize – 1**
- The function $f(i) = i*h_p(r)$ satisfies Property 2 provided $h_p(r)$ and tableSize are relatively prime.
- To guarantee Property 2, tableSize must be a prime number.
- Common definitions for $\mathbf{h_p}$ are :
  - $h_p(key) = 1 + key \% (tableSize - 1)$
  - $h_p(key) = q - (key \% q)$ where **q** is a prime less than **tableSize**
  - $h_p(key) = q*(key \% q)$ where **q** is a prime less than **tableSize**

Open Addressing: Double Hashing

Performance of Double hashing:

- Much better than linear or quadratic probing because it eliminates both primary and secondary clustering.
- BUT requires a computation of a second hash function $\mathbf{h_p}$.

Example: Load the keys **18, 26, 35, 9, 26, 47, 96, 36, and 70** in this order, in an empty hash table of size **13**

$$h_i(key) = [h(key) + i*h_p(key)]\% \text{ tableSize}$$

(a) using double hashing with the first hash function: **h(key) = key % 13** and the second hash function: **h_p(key) = 1 + key % 12**

(b) using double hashing with the first hash function: **h(key) = key % 13** and the second hash function: **h_p(key) = 7 - key % 7**

Show all computations.

Solution in class At board

Open Addressing: Double Hashing

Example: Load the keys **18, 26, 35, 9, 26, 47, 96, 36, and 70** in this order, in an empty hash table of size **13**

$$h_i(key) = [h(key) + i*h_p(key)]\% \text{ tableSize}$$

(a) using double hashing with the first hash function: **h(key) = key % 13** and the second hash function: **h_p(key) = 1 + key % 12**

Open Addressing: Double Hashing

Load the keys **18, 26, 35, 9, 64, 47, 96,36,and 70** , in this order, in a hash table of size **13** using double hashing with **h(x) = 1+x%tablesize-1** and the hash function

$h_0(18) = (18\%13)\%13 = 5$

$h_0(26) = (26\%13)\%13 = 0$

$h_0(35) = (35\%13)\%13 = 9$

$h_0(9) = (9\%13)\%13 = 9$ collision

$h_p(9) = 1 + 9\%12 = 10$

$h_1(9) = (9 + 1*10)\%13 = 6$

$h_0(64) = (64\%13)\%13 = 12$

$h_0(47) = (47\%13)\%13 = 8$

$h_0(96) = (96\%13)\%13 = 5$ collision

$h_p(96) = 1 + 96\%12 = 1$

$h_1(96) = (5 + 1*1)\%13 = 6$ collision

$h_2(96) = (5 + 2*1)\%13 = 7$

$h_0(36) = (36\%13)\%13 = 10$

$h_0(70) = (70\%13)\%13 = 5$ collision

$h_p(70) = 1 + 70\%12 = 11$

$h_1(70) = (5 + 1*11)\%13 = 3$

$$h_i(key) = [h(key) + i*h_p(key)]\% 13$$

$$h(key) = key \% 13$$

$$h_p(key) = 1 + key \% 12$$

0	1	2	3	4	5	6	7	8	9	10	11
26			70		18	9	96	47	35	36	

Open Addressing: Double Hashing

BIRZEIT UNIVERSITY

$h_0(18) = (18\%13)\%13 = 5$
$h_0(26) = (26\%13)\%13 = 0$
$h_0(35) = (35\%13)\%13 = 9$
$h_0(9) = (9\%13)\%13 = 9$ collision
  $h_p(9) = 7 - 9\%7 = 5$
  $h_1(9) = (9 + 1*5)\%13 = 1$
$h_0(64) = (64\%13)\%13 = 12$
$h_0(47) = (47\%13)\%13 = 8$
$h_0(96) = (96\%13)\%13 = 5$ collision
  $h_p(96) = 7 - 96\%7 = 2$
  $h_1(96) = (5 + 1*2)\%13 = 7$
$h_0(36) = (36\%13)\%13 = 10$
$h_0(70) = (70\%13)\%13 = 5$ collision
  $h_p(70) = 7 - 70\%7 = 7$
  $h_1(70) = (5 + 1*7)\%13 = 12$ collision
  $h_2(70) = (5 + 2*7)\%13 = 6$

$h_i(key) = [h(key) + i*h_p(key)]\% 13$

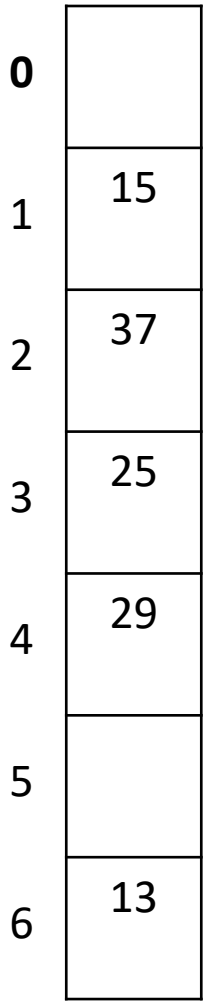
$h(key) = key \% 13$

$h_p(key) = 7 - key \% 7$

0	1	2	3	4	5	6	7	8	9	10	11	12
26	9				18	70	96	47	35	36		64

Exercise: If the hash table after using linear probing is as shown below?

a. Insert 29?
b. Is there a problem(s) is hashing table? If there is state it and give a solution (s)



Solution in class, At board

*.problems Collision and load factor more than 70%

*. Solutions is Rehashing (done at board)

Rehashing

- As noted before, with open addressing, if the hash tables become too full, performance can suffer a lot.
- So, what can we do?
- We can double the hash table size, modify the hash function, and re-insert the data.
  - More specifically, the new size of the table will be the **first prime that is more than twice as large** as the old table size.

newTable= prime>2*oldSize

When to Rehash?

- When first insertion failed
- The table is half full → load factor 50%
- Load factor = 75%

Open Addressing : pros and cons

- **Advantages of Open addressing:**
  1. All items are stored in the hash table itself. There is no need for another data structure.
  2. Open addressing is more efficient storage-wise.

- **Disadvantages of Open Addressing:**
  1) The keys of the objects to be hashed must be distinct.
  2) Dependent on choosing a proper table size.
  3) Requires the use of a three-state (Occupied, Empty, or Deleted) flag in each cell.

Separate Chaining : pros and cons

Advantages:

1. **Collision** resolution is **simple** and efficient.
2. The hash table can hold more elements without the large performance deterioration of open addressing (The l**oad factor can be 1 or greater**)
3. **Deletion is easy** - no special flag values are necessary.
4. **Table size** need not be a **prime number.**

Disadvantages:

1. It requires the **implementation** of a separate data structure for chains, and code to manage it.
2. The main cost of chaining is the **extra space** required for the **linked lists.**
3. For some **languages, creating new nodes** (for linked lists) is **expensive** and slows down the system.

Separate Chaining vs. Open-addressing

	Separate Chaining	Open Addressing
1.	Chaining is Simpler to implement.	Open Addressing requires more computation.
2.	In chaining, Hash table never fills up, we can always add more elements to chain.	In open addressing, table may become full.
3.	Chaining is Less sensitive to the hash function or load factors.	Open addressing requires extra care for to avoid clustering and load factor.
4.	Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.	Open addressing is used when the frequency and number of keys is known.
5.	Cache performance of chaining is not good as keys are stored using linked list.	Open addressing provides better cache performance as everything is stored in the same table.
6.	Wastage of Space (Some Parts of hash table in chaining are never used).	In Open addressing, a slot can be used even if an input doesn't map to it.
7.	Chaining uses extra space for links.	No links in Open addressing

Separate Chaining with String Keys

- Recall that search keys can be numbers, strings or some other object.
- A hash function for a string s = c0,c1,c2,…cn-1 can be defined as:

  hash = $(c_0 + c_1 + c_2 + \ldots + c_{n-1})$ % tableSize

  this can be implemented as:

```
typedef unsigned int INDEX

INDEX hash ( char *key, unsigned int H_SIZE)
    {
     unsigned int hash_val = 0;
           while ( *key != '\0')
           hash_val += *key++;
      return ( hash_val % H_SIZE);
    }
```

tea,ate
(TableSize =10,007 prime) ASCII code at most 127char
with one word has 8 char lenght =127 * 8= {0,…,1016}

→ **This not equitable distribution**

Separate Chaining with String Keys

- Alternative hash functions for a string

  $s = c_0c_1c_2...c_{n-1}$

exist, some are:

$$hash = (c_0 + 27 * c_1 + 729 * c_2) \% tableSize$$

```
INDEX hash ( char *key, unsigned int Table_SIZE)
  {
      return (( key[0] + 27 * key[1] + 729 * key[2] ) %  Table_SIZE );
  }
```

```
tea, ate, fashion , fashionable
TableSize =10,007 prime, examine first 3
  characters, (26 char + NULL=27)
```

→ **This not equitable distribution, since the letters are not random distributed**

$$\text{hash} = \sum_{i=0}^{KeySize\ i-1} Key[KeySize - i - 1] \cdot 32^{i}$$

tea

```
 INDEX hash ( char *key, unsigned int TableSIZE)
{
     unsigned int hash_val = 0;
       while ( *key != '\0')
        hash_val = (hash_val << 5 ) + *key++;
   if( hashVal < 0 )
     hashVal += tableSize;
     return ( hash_val % H_SIZE);
}
```

00000011, 00000110 , 00001100, 00011000, 00110000

THANK YOU