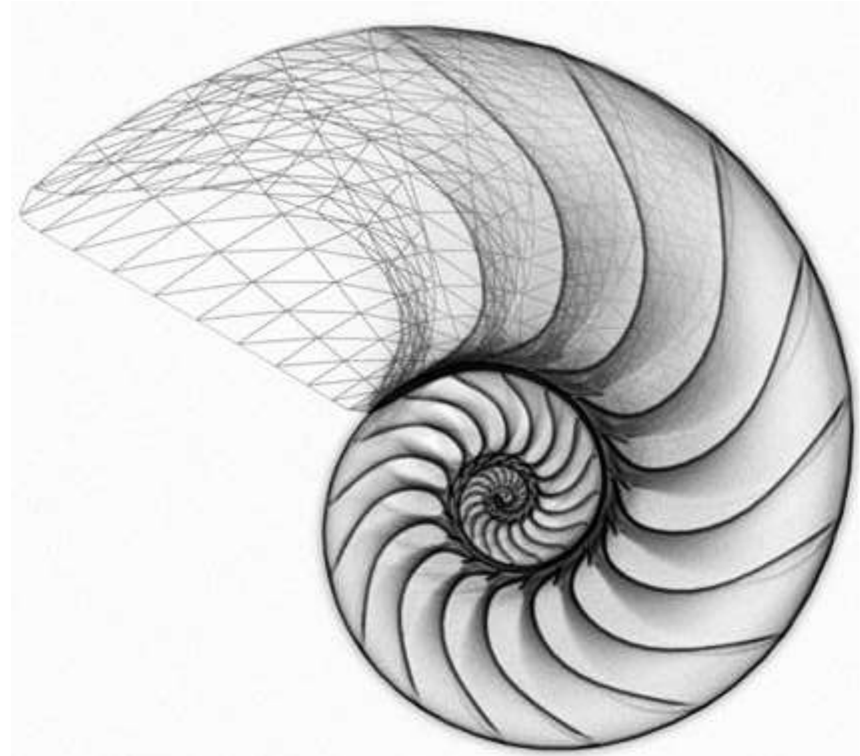


Hashing

CSE225: Data Structures and Algorithms

Introduction to Hashing

- Suppose that we want to store 10,000 students records (each with a 5-digit ID) in a given container.

  · A linked list implementation would take `O(n)` time.

  · A height balanced tree would give `O(log n)` access time.

  · Using an array of size 100,000 would give `O(1)` access time but will lead to a lot of space wastage.

- Is there some way that we could get `O(1)` access without wasting a lot of space?

- The answer is hashing.

Hash Functions

- A ***hash function***, **`h`**, is a function which transforms a key from a set, **`K`**, into an index in a table of size n:

  `h: K -> {0, 1, ..., n-2, n-1}`

  - A key can be a number, a string, a record etc.
  - The size of the set of keys, **`|K|`**, to be relatively very large.
  - It is possible for different keys to hash to the same array location.
    - This situation is called *collision* and the colliding keys are called *synonyms*.

Example 1: Illustrating Hashing

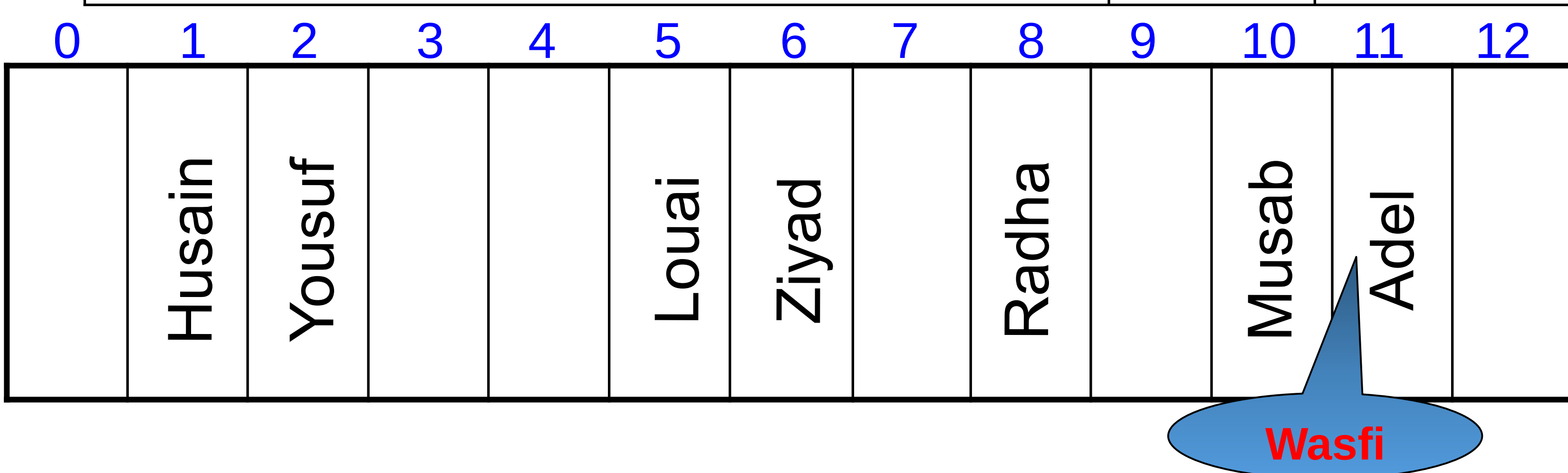
- Use the function `f(r) = r.id % 13` to load the following

  records into an array of size 13.

Al-Otaibi, Ziyad	1.73	985926
Al-Turki, Musab Ahmad Bakeer	1.60	970876
Al-Saegh, Radha Mahdi	1.58	980962
Al-Shahrani, Adel Saad	1.80	986074
Al-Awami, Louai Adnan Muhammad	1.73	970728
Al-Amer, Yousuf Jauwad	1.66	994593
Al-Helal, Husain Ali AbdulMohsen	1.70	996321
Al-Khatib, Wasfi Ghassan	1.74	863523

Example 1: Introduction to Hashing (cont'd)

Name	ID	h(r) = id % 13
Al-Otaibi, Ziyad	985926	6
Al-Turki, Musab Ahmad Bakeer	970876	10
Al-Saegh, Radha Mahdi	980962	8
Al-Shahrani, Adel Saad	986074	11
Al-Awami, Louai Adnan Muhammad	970728	5
Al-Amer, Yousuf Jauwad	994593	2
Al-Helal, Husain Ali AbdulMohsen	996321	1
Al-Khatib, Wasfi Ghassan	863523	11



Hash Tables

- There are two types of Hash Tables: **Open-addressed Hash Tables** and **Separate-Chained Hash Tables**.

- **An Open-addressed *Hash Table*** is a one-dimensional array indexed by integer values that are computed by an index function called a ***hash function***.

- **A Separate-Chained *Hash Table*** is a one-dimensional array of linked lists indexed by integer values that are computed by an index function called a ***hash function***.

- Hash tables are sometimes referred to as ***scatter tables***.

- Typical hash table operations are:
    - ***Initialization.***
    - ***Insertion.***
    - ***Searching***
    - ***Deletion.***

Types of Hashing

- There are two types of hashing :

  1. ***Static hashing***: In static hashing, the hash function maps search-key values to a fixed set of *locations*.

  2. ***Dynamic hashing***: In dynamic hashing a hash table can grow to handle more items. The associated hash function must change as the table grows.

- The ***load factor*** of a hash table is the ratio of the number of keys in the table to the size of the hash table.
  - What do you think will happen when the load factor becomes high?
  - With open addressing, the load factor cannot exceed 1. With chaining, the load factor often exceeds 1.

Desired Properties of Hash Functions

- A good hash function should:
    - ***Minimize*** collisions.
    - Be ***easy*** and ***quick*** to compute.
    - Distribute key values ***evenly*** in the hash table.
    - Use ***all the information*** provided in the key.

Common Hashing Functions

1. **Division Remainder (using the table size as the divisor)**

- Computes hash value from key using the % operator.

- Table size that is a power of 2 like 32 and 1024 should be avoided, for it leads to more collisions.

- Also, powers of 10 are not good for table sizes when the keys rely on decimal integers.

- Prime numbers not close to powers of 2 are better table size values.

Common Hashing Functions (cont'd)

2. Folding

- It involves splitting keys into two or more parts and then combining the parts to form the hash addresses.

- To map the key 25936715 to a range between 0 and 9999, we can:
  - · split the number into two as 2593 and 6715 and
  - · add these two to obtain 9308 as the hash value.

- Very useful if we have keys that are very large.

- Fast and simple especially with bit patterns.

- A great advantage is ability to transform non-integer keys into integer values.

Common Hashing Functions (cont'd)

3. Mid-Square

- The key is squared and the middle part of the result taken as the hash value.
- To map the key **3121** into a hash table of size **1000**, we square it $\mathbf{3121^2 = 9740641}$ and extract **406** as the hash value.
- Works well if the keys do not contain a lot of leading or trailing zeros.
- Non-integer keys have to be preprocessed to obtain corresponding integer values.

Common Hashing Functions (cont'd)

4. Truncation or Digit/Character Extraction

- Works based on the distribution of digits or characters in the key.

- More evenly distributed digit positions are extracted and used for hashing purposes.

- For instance, students IDs or ISBN codes may contain common subsequences which may increase the likelihood of collision.

- Very fast but digits/characters distribution in keys may not be very even.

Common Hashing Functions (cont’d)

5. Radix Conversion

- Transforms a key into another number base to obtain the hash value.
- Typically use number base other than base 10 and base 2 to calculate the hash addresses.
- To map the key 55354 in the range 0 to 9999 using base 11 we have:

$$55354_{10} = 38652_{11}$$

- We may truncate the high-order 3 to yield 8652 as our hash address within 0 to 9999.

Common Hashing Functions (cont’d)

6. Use of a Random-Number Generator

- Given a seed as parameter, the method generates a random number.

- The algorithm must ensure that:

    - It always generates the same random value for a given key.

    - It is unlikely for two keys to yield the same random value.

- The random number produced can be transformed to produce a valid hash value.

Some Applications of Hash Tables

- **Database systems**: Specifically, those that require efficient random access. Generally, database systems try to optimize between two types of access methods: sequential and random. Hash tables are an important part of efficient random access because they provide a way to locate data in a constant amount of time.

- **Symbol tables**: The tables used by compilers to maintain information about symbols from a program. Compilers access information about symbols frequently. Therefore, it is important that symbol tables be implemented very efficiently.

- **Data dictionaries**: Data structures that support adding, deleting, and searching for data. Although the operations of a hash table and a data dictionary are similar, other data structures may be used to implement data dictionaries. Using a hash table is particularly efficient.

- **Network processing algorithms**: Hash tables are fundamental components of several network processing algorithms and applications, including route lookup, packet classification, and network monitoring.

- **Browser Cashes**: Hash tables are used to implement browser cashes.

Problems for Which Hash Tables are not Suitable

1. **Problems for which data ordering is required.**
   Because a hash table is an unordered data structure, certain operations are difficult and expensive. Range queries, proximity queries, selection, and sorted traversals are possible only if the keys are copied into a sorted data structure. There are hash table implementations that keep the keys in order, but they are far from efficient.

2. **Problems having multidimensional data.**

3. **Prefix searching** especially if the keys are long and of variable-lengths.

4. **Problems that have dynamic data**:
   Open-addressed hash tables are based on 1D-arrays, which are difficult to resize once they have been allocated. Unless you want to implement the table as a dynamic array and rehash all of the keys whenever the size changes. This is an incredibly expensive operation. An alternative is use a separate-chained hash tables or dynamic hashing.

5. **Problems in which the data does not have unique keys.**
   Open-addressed hash tables cannot be used if the data does not have unique keys. An alternative is use separate-chained hash tables.

Collision Resolution Techniques

- There are two broad ways of collision resolution:

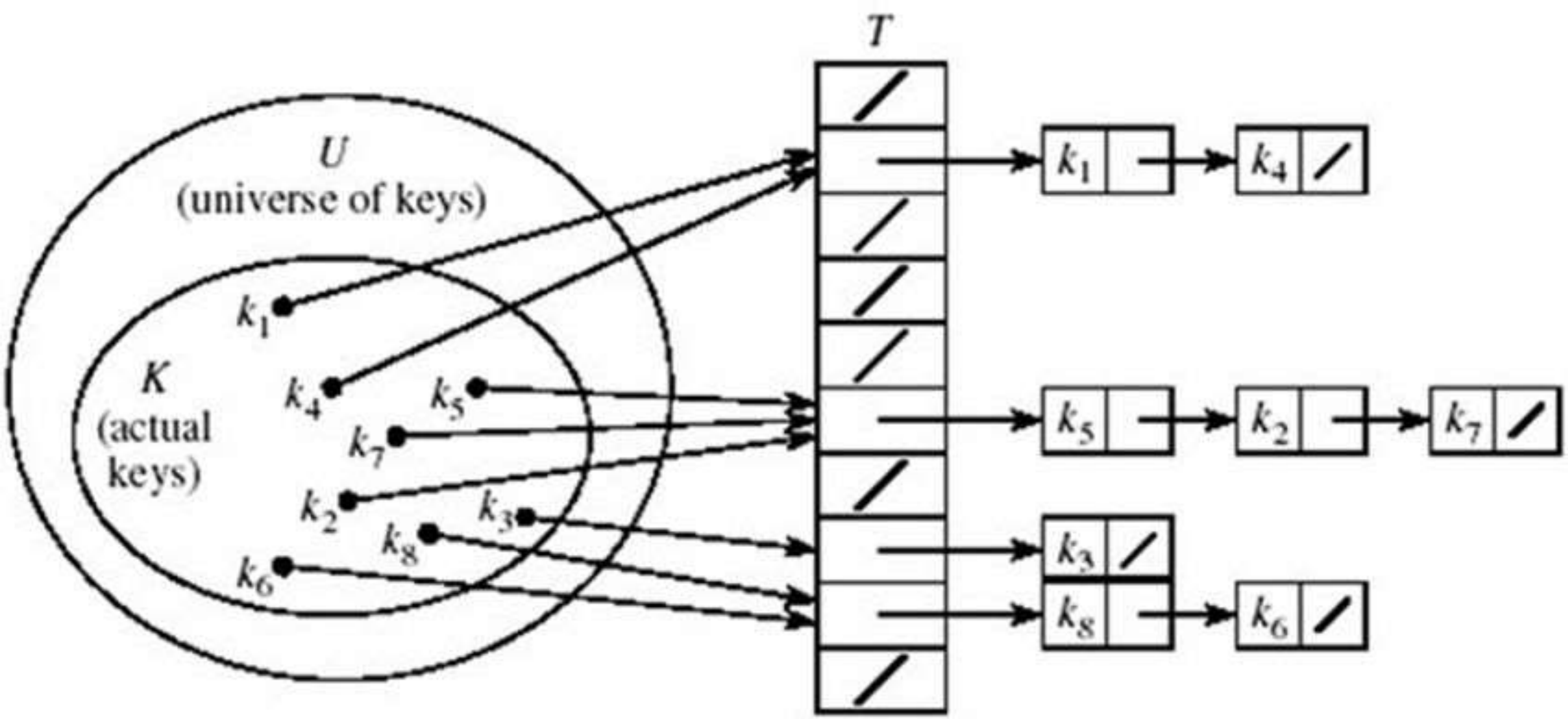
1. Separate Chaining:: An array of linked list implementation.

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

- 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.



Separate Chaining (cont'd)

- 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`.
- **Example:** Load the keys **23, 13, 21, 14, 7, 8, and 15** , in this order, in a hash table of size **7** using separate chaining with the hash function: **h(key) = key % 7**

h(23) = 23 % 7 = 2

h(13) = 13 % 7 = 6

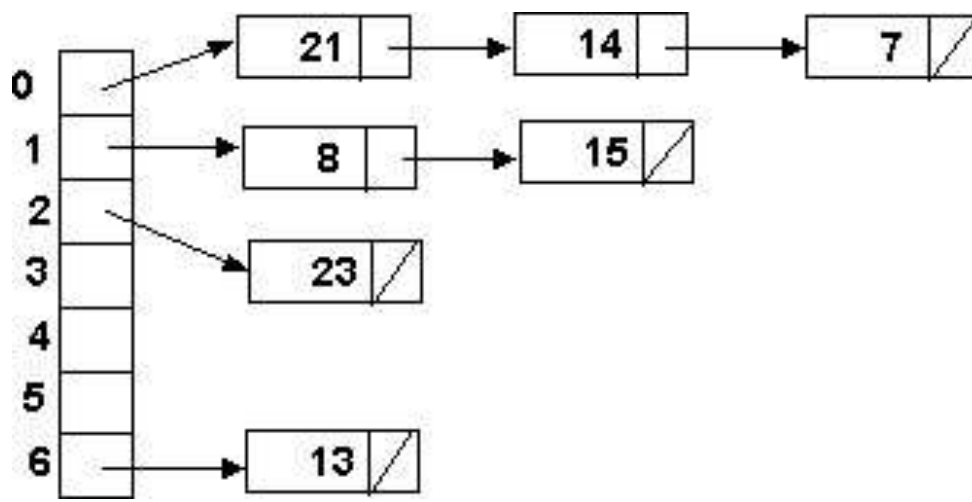
h(21) = 21 % 7 = 0

h(14) = 14 % 7 = 0 collision

h(7) = 7 % 7 = 0 collision

h(8) = 8 % 7 = 1

h(15) = 15 % 7 = 1 collision



Introduction to 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.
- The most common probe sequences are of the form:

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

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

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

Property 1: c(0) = 0

Property 2: The set of values **{c(0) % n, c(1) % n, c(2) % n, . . . , c(n-1) % n}** must be a permutation of **{0, 1, 2,. . ., n – 1}**, that is, it must contain every integer between **0** and **n - 1** inclusive.

Introduction to Open Addressing (cont'd)

- The function **c(i)** is used to resolve collisions.

- 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), ..., h_{n-1}(r)** are examined until an empty slot is found.
  - If the probe sequence contains **one or more deleted cells**, the key is inserted into the **first deleted cell**, otherwise, it is inserted into the **empty cell.**

- 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 **c(i)**.

- Common definitions of **c(i)** are:

Collision resolution technique	c(i)
Linear probing	i
Quadratic probing	$\pm i^2$
Double hashing	$i*h_p(key)$

where **h_p(key)** is another hash function.

Introduction to Open Addressing (cont'd)

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

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

Open Addressing: Linear Probing

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

  $c(i) = i$ **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**

- For **c(i) = a*i** to satisfy Property 2, **a** and **tableSize** must be relatively prime.

Linear Probing Example

Example: Perform the operations given below, in the given order, on an initially empty hash table of size **13** using linear probing with **c(i) = i** and the hash function: **h(key) = key % 13**:

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

- The required probe sequences are given by:

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

Linear Probing Example (cont'd)

Initial state of the hash table:

Index	Status	Value
0	E	
1	E	
2	E	
3	E	
4	E	
5	E	
6	E	
7	E	
8	E	
9	E	
10	E	
11	E	
12	E	

Linear Probing Example (cont'd)

$h(key) = key \% 13$

$h(18) = 18 \% 13$

$= 5$

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

$h_o(18) = (5 + 0) \% 13$

$= 5 \% 13$

$= 5$

insert(18)

Index	Status	Value
0	E	
1	E	
2	E	
3	E	
4	E	
5	O	18
6	E	
7	E	
8	E	
9	E	
10	E	
11	E	
12	E	

Linear Probing Example (cont'd)

h(key) = key % 13

h(26) = 26 % 13

= 0

h_i(key) = (h(key) + i) % 13

$h_o(26) = (0 + 0) \% 13$

$= 0 \% 13$

$= 0$

insert(26)

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

Linear Probing Example (cont'd)

$h(key) = key \% 13$

insert(35)

$h(35) = 35 \% 13$

$= 9$

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

$h_o(26) = (9 + 0) \% 13$

$= 9 \% 13$

$= 9$

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

Linear Probing Example (cont'd)

h(key) = key % 13

insert(9)

$h(9) = 9 \% 13$

$= 9$

h_i(key) = (h(key) + i) % 13

$h_o(9) = (9 + 0) \% 13$

$= 9 \% 13$

$= 9$ COLLISION

$h_1(9) = (9 + 1) \% 13$

$= 10 \% 13$

$= 10$

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

Linear Probing Example (cont'd)

find(15)

h(key) = key % 13

$h(15) = 15 \% 13$
$= 2$

h_i(key) = (h(key) + i) % 13

$h_o(15) = (2 + 0) \% 13$

$= 2 \% 13$

$= 2$ search FAILS because cell 2 has EMPTY status

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

Linear Probing Example (cont'd)

find(48)

h(key) = key % 13

$h(48) = 48 \% 13$
$= 9$

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

$h_o(48) = (9 + 0) \% 13 = 9 \% 13 = 9$
continue search, because cell 9 has OCUPIED status

$h_1(48) = (9 + 1) \% 13 = 10 \% 13 = 10$
continue search, because cell 10 has OCUPIED status

$h_2(48) = (9 + 2) \% 13 = 11 \% 13 = 11$
search FAILS, because cell 11 has EMPTY status

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

Linear Probing Example (cont'd)

delete(35)

h(key) = key % 13

h(35) = 35 % 13
= 9

h_i(key) = (h(key) + i) % 13

$h_o(35) = (9 + 0)$ % 13 = 9 % 13 = 9
35 found and deleted, cell 9 status changed to DELETED

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

Linear Probing Example (cont'd)

insert(9)

$h(key) = key \% 13$

$h(9) = 9 \% 13$
$= 9$

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

$h_0(9) = (9 + 0) \% 13 = 9 \% 13 = 9$
Cell 9 has DELETED status. CONTINUE SEARCHING

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

Cell 10 is OCCUPIED with 9 ➔ Duplicate key. Cannot insert 9

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

Linear Probing Example (cont'd)

h(key) = key % 13

delete(40)

$h(40) = 40 \% 13$
$= 1$

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

$h_o(40) = (1 + 0) \% 13 = 1 \% 13 = 1$

cell 1 has EMPY status, 40 not found, deletion FAILS

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

Linear Probing Example (cont'd)

find(9)

$h(key) = key \% 13$

$h(9) = 9 \% 13$
$= 9$

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

$h_o(9) = (9 + 0) \% 13 = 9 \% 13 = 9$
Continue search, because cell 9 has DELETED status.

$h_1(9) = (9 + 1) \% 13 = 10 \% 13 = 10$
9 found at cell 10.

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

Linear Probing Example (cont'd)

insert(64)

$h(key) = key \% 13$

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

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

$h_o(64) = (12 + 0) \% 13 = 12 \% 13 = 12$

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

Linear Probing Example (cont'd)

insert(47)

h(key) = key % 13

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

h_i(key) = (h(key) + i) % 13

$h_o(47) = (8 + 0) \% 13 = 8 \% 13 = 8$

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	E	
12	O	64

Linear Probing Example (cont'd)

find(35)

h(key) = key % 13

h(35) = 35 % 13
= 9

h_i(key) = (h(key) + i) % 13

$h_0(35) = (9 + 0) \% 13 = 9 \% 13 = 9$
Search continues because cell 9 has DELETED status.

$h_1(35) = (9 + 1) \% 13 = 10 \% 13 = 10$
Search continues because cell 10 has OCUPPIED status and key is not 35.

$h_2(35) = (9 + 2) \% 13 = 11 \% 13 = 11$
Search FAILS because cell 11 has EMPTY status.

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	E	
12	O	64

Linear Probing Example (cont'd)

insert(77)

$h(key) = key \% 13$

$h(77) = 77 \% 13$
$= 12$

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

$h_o(77) = (12 + 0) \% 13 = 12 \% 13 = 12$ collision

$h_1(77) = (12 + 1) \% 13 = 13 \% 13 = 0$ collision

$h_2(77) = (12 + 2) \% 13 = 14 \% 13 = 1$

Index	Status	Value
0	O	26
1	O	77
2	E	
3	E	
4	E	
5	O	18
6	E	
7	E	
8	O	47
9	D	35
10	O	9
11	E	
12	O	64

Linear Probing Example (cont'd)

$h(key) = key \% 13$

insert(21)

$h(21) = 21 \% 13$
$= 8$

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

$h_o(21) = (8 + 0) \% 13 = 8 \% 13 = 8$ collision

$h_1(21) = (8 + 1) \% 13 = 9 \% 13 = 9$ Cell 9 has DELETED status, Search continues.

$h_2(21) = (8 + 2) \% 13 = 10 \% 13 = 10$
Cell 10 is OCCUPIED with a different value, search continues.

$h_3(21) = (8 + 3) \% 13 = 11 \% 13 = 11$
Cell 11 is empty, 21 is inserted in cell 9

Index	Status	Value
0	O	26
1	O	77
2	E	
3	E	
4	E	
5	O	18
6	E	
7	E	
8	O	47
9	O	21
10	O	9
11	E	
12	O	64

Linear Probing Example (cont'd)

h(key) = key % 13

h(26) = 26 % 13
= 0

h_i(key) = (h(key) + i) % 13

$h_o(26) = (0 + 0) \% 13 = 0 \% 13 = 0$

26 is deleted and status changed to DELETED.

delete(26)

Index	Status	Value
0	D	26
1	O	77
2	E	
3	E	
4	E	
5	O	18
6	E	
7	E	
8	O	47
9	O	21
10	O	9
11	E	
12	O	64

Linear Probing Example (cont'd)

insert(39)

h(key) = key % 13

h(39) = 39 % 13
= 0

h_i(key) = (h(key) + i) % 13

$h_o(39) = (0 + 0) \% 13 = 0 \% 13 = 0$

Status of cell 0 is DELETED, search continues.

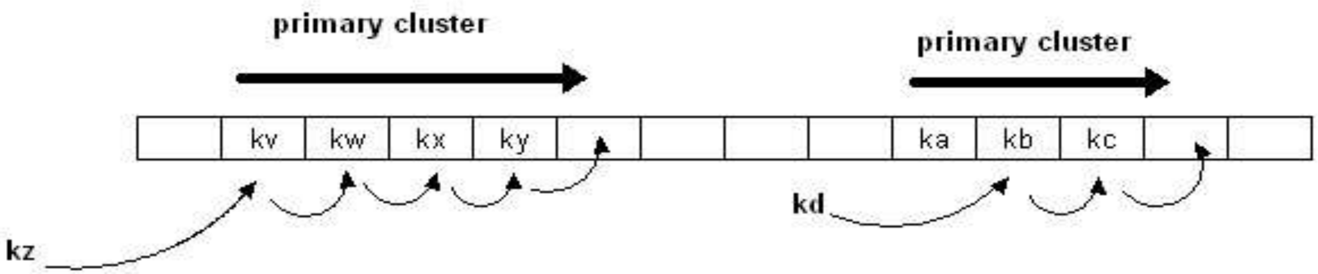
$h_1(39) = (0 + 1) \% 13 = 1 \% 13 = 1$
Cell 1 is OCCUPIED with a different value.

$h_2(39) = (0 + 2) \% 13 = 2 \% 13 = 2$
Cell 2 status is EMPTY, hence 39 is inserted in cell 0

Index	Status	Value
0	O	39
1	O	77
2	E	
3	E	
4	E	
5	O	18
6	E	
7	E	
8	O	47
9	O	21
10	O	9
11	E	
12	O	64

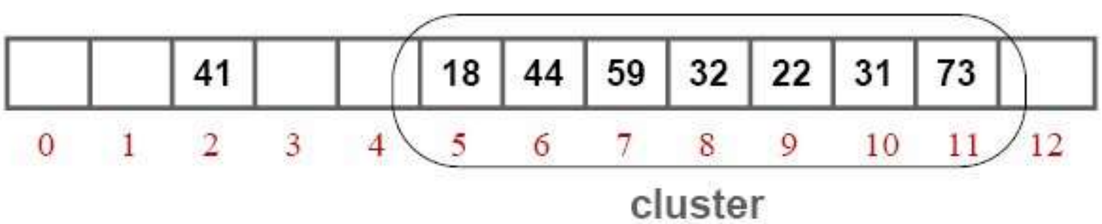
Disadvantage of Linear Probing: Primary Clustering

- 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 **c(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+1a



Open Addressing: Quadratic Probing

- Quadratic probing eliminates primary clusters.
- **c(i)** is a quadratic function in **i** of the form **$c(i) = a*i^2 + b*i$**. Usually **c(i)** is chosen as:

  $c(i) = \pm i^2$ **for i = 0, 1, . . . , (tableSize – 1) / 2**

- The probe sequences are then given by:

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

- Note for Quadratic Probing:
  - Hashtable size should not be an even number; otherwise Property 2 will not be satisfied.
  - Ideally, table size should be a prime of the form 4j+3, where j is an integer. This choice of table size guarantees Property 2.

Open Addressing: Quadratic Probing

- For $\mathbf{h_0(key) = [h(key) \pm i^2] \% tableSize}$ the probe sequence starts at $\mathbf{h_0(key) = h(key)}$ and then it examines cells 1, -1, 4, -4, 9, -9 and so on, away from the original probe. For this probe sequence, normalization is done when a **negative** index is computed:
  normalizedIndex = (computedIndex + tableSize) % tableSize

- Example: Load the keys **23, 13, 21, 14, 7, 8, and 15**, in this order, in a hash table of size **7** using quadratic probing with $\mathbf{c(i) = \pm i^2}$ and the hash function: **h(key) = key % 7**
- The required probe sequences are given by:

$$h_i(key) = (h(key) \pm i^2) \% 7 \quad i = 0, 1, 2, 3$$

Quadratic Probing (cont'd)

$h_i(key) = (h(key) \pm i^2) \% 7 \quad i = 0, 1, 2, 3$

$h_0(23) = (23 \% 7) \% 7 = 2$

$h_0(13) = (13 \% 7) \% 7 = 6$

$h_0(21) = (21 \% 7) \% 7 = 0$

$h_0(14) = (14 \% 7) \% 7 = 0$ **collision**

$h_1(14) = (0 + 1^2) \% 7 = 1$

$h_0(7) = (7 \% 7) \% 7 = 0$ **collision**

$h_1(7) = (0 + 1^2) \% 7 = 1$ **collision**

$h_{-1}(7) = (0 - 1^2) \% 7 = -1$

NORMALIZE: $(-1 + 7) \% 7 = 6$ **collision**

$h_2(7) = (0 + 2^2) \% 7 = 4$

$h_0(8) = (8 \% 7)\%7 = 1$ **collision**

$h_1(8) = (1 + 1^2) \% 7 = 2$ **collision**

$h_{-1}(8) = (1 - 1^2) \% 7 = 0$ **collision**

$h_2(8) = (1 + 2^2) \% 7 = 5$

$h_0(15) = (15 \% 7)\%7 = 1$ **collision**

$h_1(15) = (1 + 1^2) \% 7 = 2$ **collision**

$h_{-1}(15) = (1 - 1^2) \% 7 = 0$ **collision**

$h_2(15) = (1 + 2^2) \% 7 = 5$ **collision**

$h_{-2}(15) = (1 - 2^2) \% 7 = -3$

NORMALIZE: $(-3 + 7) \% 7 = 4$ **collision**

$h_3(15) = (1 + 3^2)\%7 = 3$

0	O	21
1	O	14
2	O	23
3	O	15
4	O	7
5	O	8
6	O	13

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 **c(i) = i²**. Assuming that each of the keys hashes to the same array index **x**. A secondary cluster will develop and grow in size: