

# 10.1 Hash tables

## Hash table overview

A **hash table** is a data structure that stores unordered items by mapping (or hashing) each item to a location in an array (or vector). Ex: Given an array with indices 0..9 to store integers from 0..500, the modulo (remainder) operator can be used to map 25 to index 5 ( $25 \% 10 = 5$ ), and 149 to index 9 ( $149 \% 10 = 9$ ). A hash table's main advantage is that searching (or inserting / removing) an item may require only  $O(1)$ , in contrast to  $O(N)$  for searching a list or to  $O(\log N)$  for binary search.

In a hash table, an item's **key** is the value used to map to an index. For all items that might possibly be stored in the hash table, every key is ideally unique, so that the hash table's algorithms can search for a specific item by that key.

Each hash table array element is called a **bucket**. A **hash function** computes a bucket index from the item's key.

PARTICIPATION ACTIVITY

10.1.1: Hash table data structure.



### Animation captions:

1. A new hash table named playerNums with 10 buckets is created. A hash function maps an item's key to the bucket index.
2. A good hash function will distribute items into different buckets.
3. Hash tables provide fast search, using as few as one comparison.

PARTICIPATION ACTIVITY

10.1.2: Hash tables.



1) A 100 element hash table has 100 \_\_\_\_\_.

- items
- buckets



2) A hash function computes a bucket index from an item's \_\_\_\_\_.

- integer value
- key

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEPACS2302ValeraFall2023





- 3) For a well-designed hash table,  
searching requires \_\_\_\_\_ on average.
- O(1)
  - O(N)
  - O(log N)

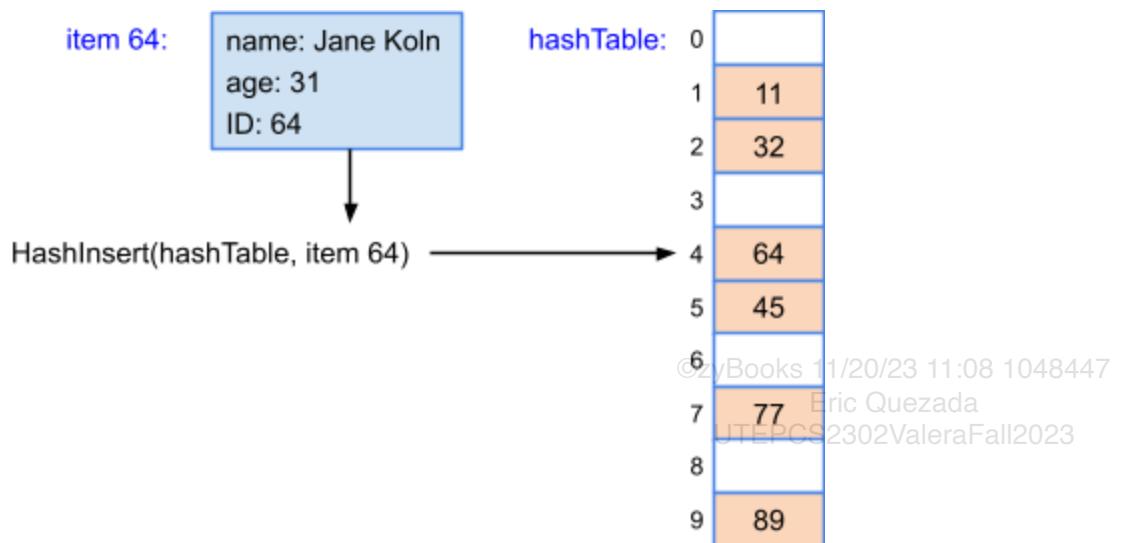
- 4) A company will store all employees in a hash table. Each employee item consists of a name, department, and employee ID number. Which is the most appropriate key?

- Name
- Department
- Employee ID number

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302-ValeraFall2023

## Item representation

Normally, each item being stored is an object with several fields, such as a person object having a name, age, and ID number, with the ID number used as the key. For simplicity, this section represents an item just by the item's key. Ex: Item 64 represents a person object with a key of 64, which is the person's ID. HashInsert(hashTable, item 64) inserts item 64 in bucket 4, representing item 64 in the hash table just by the key 64.



## Hash table operations

A common hash function uses the **modulo operator %**, which computes the integer remainder when dividing two numbers. Ex: For a 20 element hash table, a hash function of key % 20 will map keys to bucket indices 0 to 19.

A hash table's operations of insert, remove, and search each use the hash function to determine an item's bucket. Ex: Inserting 113 first determines the bucket to be  $113 \% 10 = 3$ .

**PARTICIPATION ACTIVITY**

## 10.1.3: Hash tables.

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEPACS2302ValeraFall2023

- 1) A modulo hash function for a 50 entry hash table is: key % \_\_\_\_\_

  
**Check****Show answer**

- 2) key % 1000 maps to indices 0 to \_\_\_\_\_

  
**Check****Show answer**

- 3) A modulo hash function is used to map to indices 0 to 9. The hash function should be: key % \_\_\_\_\_

  
**Check****Show answer**

- 4) Given a hash table with 100 buckets and modulo hash function, in which bucket will HashInsert(table, item 334) insert item 334?

  
**Check****Show answer**

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEPACS2302ValeraFall2023



- 5) Given a hash table with 50 buckets and modulo hash function, in which bucket will HashSearch(table, 201) search for the item?

  
**Check****Show answer**

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023

**PARTICIPATION ACTIVITY****10.1.4: Hash table search efficiency.**

Consider a modulo hash function (key % 10) and the following hash table.

numsTable:	0	
	11	
	22	
	45	
	47	
	39	

- 1) How many buckets will be checked for HashSearch(numsTable, 45)?
- 1
  - 6
  - 10



- 2) If item keys range from 0 to 49, how many keys may map to the same bucket?

- 1
- 5
- 50

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEPACS2302ValeraFall2023



3) If a linear search were applied to the array, how many array elements would be checked to find item 45?

- 1
- 6
- 10

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023

## Empty cells

The approach for a hash table algorithm determining whether a cell is empty depends on the implementation. For example, if items are simply non-negative integers, empty can be represented as -1. More commonly, items are each an object with multiple fields (name, age, etc.), in which case each hash table array element may be a pointer. Using pointers, empty can be represented as null.

## Collisions

A **collision** occurs when an item being inserted into a hash table maps to the same bucket as an existing item in the hash table. Ex: For a hash function of key % 10, 55 would be inserted in bucket 55 % 10 = 5; later inserting 75 would yield a collision because 75 % 10 is also 5. Various techniques are used to handle collisions during insertions, such as chaining or open addressing. **Chaining** is a collision resolution technique where each bucket has a list of items (so bucket 5's list would become 55, 75). **Open addressing** is a collision resolution technique where collisions are resolved by looking for an empty bucket elsewhere in the table (so 75 might be stored in bucket 6). Such techniques are discussed later in this material.

### PARTICIPATION ACTIVITY

#### 10.1.5: Hash table collisions.



1) A hash table's items will be positive integers, and -1 will represent empty. A 5-bucket hash table is: -1, -1, 72, 93, -1. How many items are in the table?

- 0
- 2
- 5

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023



2) A hash table has buckets 0 to 9 and uses a hash function of key % 10. If the table is initially empty and the following inserts are applied in the order shown, the insert of which item results in a collision?

HashInsert(hashTable, item 55)

HashInsert(hashTable, item 90)

HashInsert(hashTable, item 95)

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023

- Item 55
- Item 90
- Item 95

**CHALLENGE  
ACTIVITY**

10.1.1: Hash tables with modulo hash function.



502696.2096894.qx3zqy7

Start

A hash table with non-negative integer keys has a modulo hash function of key % 15.

Hash function index range: 0 to

Item 190 will go in bucket

1

2

3

Check

Next

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023

## 10.2 Chaining

**Chaining** handles hash table collisions by using a list for each bucket, where each list may store multiple items that map to the same bucket. The insert operation first uses the item's key to determine the bucket, and then inserts the item in that bucket's list. Searching also first determines the bucket, and then searches the bucket's list. Likewise for removes.

**PARTICIPATION ACTIVITY**

10.2.1: Hash table with chaining.



©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023

**Animation content:**

undefined

**Animation captions:**

1. A hash table with chaining uses a list for each bucket. The insert operation first uses the item's key to determine the mapped bucket, and then inserts the item in that bucket's list.
2. A bucket may store multiple items with different keys that map to the same bucket. If collisions occur, items are inserted in the bucket's list.
3. Search first uses the item's key to determine the mapped bucket, and then searches the items in that bucket's list.

Figure 10.2.1: Hash table with chaining: Each bucket contains a list of items.

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEPACS2302ValeraFall2023

```

HashInsert(hashTable, item) {
    if (HashSearch(hashTable, item->key) == null) {
        bucketList = hashTable[Hash(item->key)]
        node = Allocate new linked list node
        node->next = null
        node->data = item
        ListAppend(bucketList, node)
    }
}

HashRemove(hashTable, item) {
    bucketList = hashTable[Hash(item->key)]
    itemNode = ListSearch(bucketList, item->key)
    if (itemNode is not null) {
        ListRemove(bucketList, itemNode)
    }
}

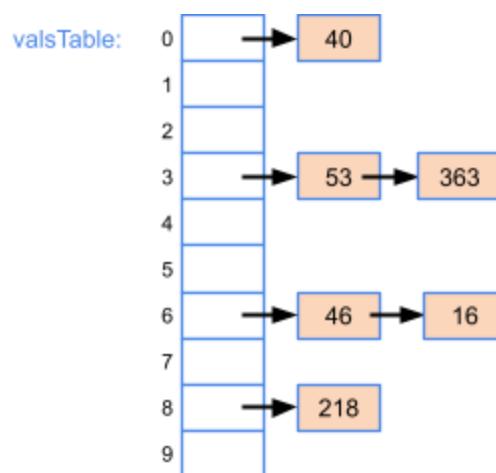
HashSearch(hashTable, key) {
    bucketList = hashTable[Hash(key)]
    itemNode = ListSearch(bucketList, key)
    if (itemNode is not null)
        return itemNode->data
    else
        return null
}

```

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEPPCS2302ValeraFall2023

**PARTICIPATION ACTIVITY**
**10.2.2: Hash table with chaining: Inserting items.**


Given hash function of key % 10, type the specified bucket's list after the indicated operation(s). Assume items are inserted at the end of a bucket's list. Type the bucket list as: 5, 7, 9 (or type: Empty).



©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEPPCS2302ValeraFall2023



1) HashInsert(valsTable, item 20)

Bucket 0's list: \_\_\_\_\_

**Check**

[Show answer](#)

2) HashInsert(valsTable, item 23)

HashInsert(valsTable, item 99)

Bucket 3's list: \_\_\_\_\_

**Check**

[Show answer](#)

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023



3) HashRemove(valsTable, 46)

Bucket 6's list: \_\_\_\_\_

**Check**

[Show answer](#)



4) HashRemove(valsTable, 218)

Bucket 8's list: \_\_\_\_\_

**Check**

[Show answer](#)

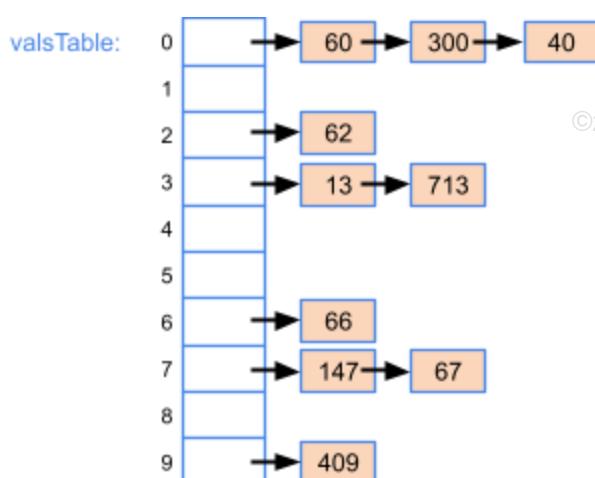


### PARTICIPATION ACTIVITY

10.2.3: Hash table with chaining: Search.



Consider the following hash table, and a hash function of key % 10.



©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEPACS2302ValeraFall2023



- 1) How many list elements are compared for  
HashSearch(valsTable, 62)?

 //**Check****Show answer**

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023

- 2) How many list elements are compared for  
HashSearch(valsTable, 40)?

 //**Check****Show answer**

- 3) What does HashSearch(valsTable, 186) return?

 //**Check****Show answer**

- 4) How many list elements are compared for  
HashSearch(valsTable, 837)?

 //**Check****Show answer****CHALLENGE ACTIVITY**

10.2.1: Chaining.



502696.2096894.qx3zqy7

**Start**

©zyBooks 11/20/23 11:08 1048447

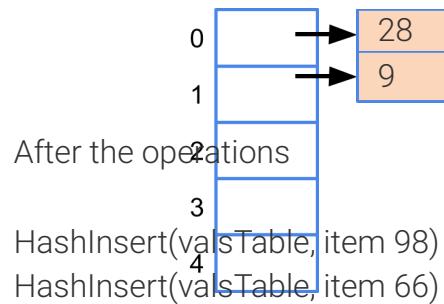
Eric Quezada

UTEPACS2302ValeraFall2023

Hash table valsTable is shown below. The hash function is key % 5. Assume items are inserted at the end of a bucket's list.

valsTable:





©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023

Select the bucket containing the following items:

- 98
- 28
- 66

1	2	3	4	5
---	---	---	---	---

**Check****Next**

## 10.3 Linear probing

### Linear probing overview

A hash table with **linear probing** handles a collision by starting at the key's mapped bucket, and then linearly searches subsequent buckets until an empty bucket is found.

**PARTICIPATION ACTIVITY**

10.3.1: Hash table with linear probing.



#### Animation captions:

- During an insert, if a bucket is not empty, a collision occurs. Using linear probing, inserts will linearly probe buckets until an empty bucket is found.
- The item is inserted in the next empty bucket.
- Search starts at the hashed location and will compare each bucket until a match is found.
- If an empty bucket is found, search returns null, indicating a matching item was not found.

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023

**PARTICIPATION ACTIVITY**

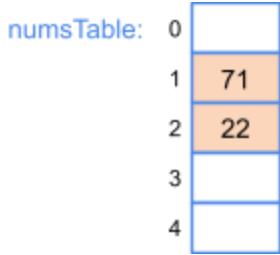
10.3.2: Hash table with linear probing: Insert.



Given hash function of key % 5, determine the insert location for each item.



## 1) HashInsert(numsTable, item 13)

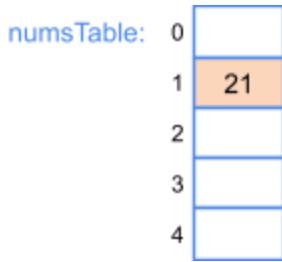


bucket =  //

**Check****Show answer**

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302-ValeraFall2023

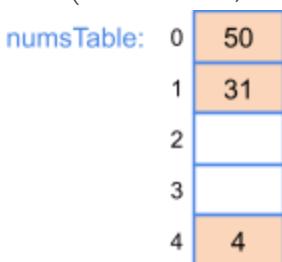
## 2) HashInsert(numsTable, item 41)



bucket =  //

**Check****Show answer**

## 3) HashInsert(numsTable, item 90)



bucket =  //

**Check****Show answer**

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302-ValeraFall2023



4) HashInsert(numsTable, item 74)

numsTable:	0	20
	1	
	2	32
	3	
	4	94

bucket =  //**Check****Show answer**

©zyBooks 11/20/23 11:08 1048447  
 Eric Quezada  
 UTEPCS2302ValeraFall2023

## Empty bucket types

Actually, linear probing distinguishes two types of empty buckets. An **empty-since-start** bucket has been empty since the hash table was created. An **empty-after-removal** bucket had an item removed that caused the bucket to now be empty. The distinction will be important during searches, since searching only stops for empty-since-start, not for empty-after-removal.

**PARTICIPATION ACTIVITY**

10.3.3: Hash with linear probing: Bucket status.



Given hash function of key % 10, determine the bucket status after the following operations have been executed.

HashInsert(valsTable, item 64)  
 HashInsert(valsTable, item 20)  
 HashInsert(valsTable, item 51)  
 HashRemove(valsTable, 51)

1) Bucket 2

- empty-since-start
- empty-after-removal



2) Bucket 1

- empty-since-start
- empty-after-removal



©zyBooks 11/20/23 11:08 1048447  
 Eric Quezada  
 UTEPCS2302ValeraFall2023

3) Bucket 4

- occupied
- empty-after-removal



## Inserts using linear probing

Using linear probing, a hash table *insert* algorithm uses the item's key to determine the initial bucket, linearly probes (or checks) each bucket, and inserts the item in the next empty bucket (the empty kind doesn't matter). If the probing reaches the last bucket, the probing continues at bucket 0. The insert algorithm returns true if the item was inserted, and returns false if all buckets are occupied.

PARTICIPATION ACTIVITY

10.3.4: Insert with linear probing.

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023



### Animation content:

undefined

### Animation captions:

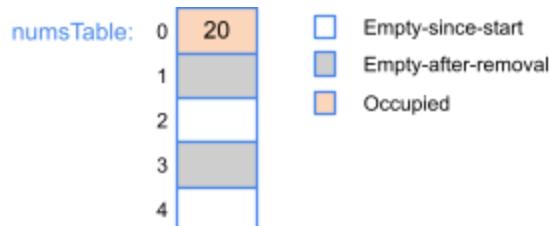
1. Insert algorithm uses the item's key to determine the initial bucket.
2. Insert linearly probes (or checks) each bucket until an empty bucket is found.
3. Item is inserted into the next empty bucket.
4. If probing reaches the last bucket without finding an empty bucket, the probing continues at bucket 0.
5. Insert linearly probes each bucket until an empty bucket is found.

PARTICIPATION ACTIVITY

10.3.5: Hash table with linear probing: Insert with empty-after-removal buckets.



For the given hash table and hash function of key % 5, what are the contents for each bucket after the following operations?



`HashInsert(numsTable, item 43)`

`HashInsert(numsTable, item 300)`

`HashInsert(numsTable, item 71)`

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023

If unable to drag and drop, refresh the page.

**Empty-since-start**

**Item 20**

**Item 300**

**Item 71**

**Item 43**

	numsTable[0]
	numsTable[1]
	numsTable[2]
	numsTable[3]
	numsTable[4]

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023

**Reset**

## Removals using linear probing

Using linear probing, a hash table *remove* algorithm uses the sought item's key to determine the initial bucket. The algorithm probes each bucket until either a matching item is found, an empty-since-start bucket is found, or all buckets have been probed. If the item is found, the item is removed, and the bucket is marked empty-after-removal.

**PARTICIPATION ACTIVITY**

10.3.6: Remove with linear probing.



### Animation content:

undefined

### Animation captions:

1. The remove algorithm uses the sought item's key to determine the initial bucket, probing buckets to find a matching item.
2. If the matching item is found, the item is removed, and the bucket is marked empty-after-removal.
3. Remove algorithm probes each bucket until either the matching item or an empty-since-start bucket is found.
4. If the matching item is found, the bucket is marked empty-after-removal.

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada

UTEPACS2302ValeraFall2023

Note that if the algorithm encounters an empty-after-removal bucket, the algorithm keeps probing, because the sought item may have been placed in a subsequent bucket before this bucket's item was removed. Ex: Removing item 42 above would start at bucket 2. Because bucket 2 is empty-after-removal, the algorithm would proceed to bucket 3, where item 42 would be found and removed.

**PARTICIPATION ACTIVITY**

## 10.3.7: Hash table with linear probing: Remove.

Consider the following hash table and a hash function of key % 10.

idsTable:	0	1	2	3	4	5	6	7	8	9
	20									
		68								
			22							
					34					
						115				
							65			
								48		
									199	

- Empty-since-start
- Empty-after-removal
- Occupied

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023

- 1) HashRemove(idsTable, 65) probes \_\_\_\_\_ buckets.

//
**Check****Show answer**

- 2) HashRemove(idsTable, 10) probes \_\_\_\_\_ buckets.

//
**Check****Show answer**

- 3) HashRemove(idsTable, 68) probes \_\_\_\_\_ buckets.

//
**Check****Show answer**

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023

**Searching using linear probing**

In linear probing, a hash table search algorithm uses the sought item's key to determine the initial bucket. The algorithm probes each bucket until either the matching item is found (returning the item), an empty-since-start bucket is found (returning null), or all buckets are probed without a match (returning null). If an empty-after-removal bucket is found, the search algorithm continues to probe the next bucket.

**Animation content:**

undefined

**Animation captions:**

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023

1. The search algorithm uses the sought item's key to determine the initial bucket, and then linearly probes each bucket until a matching item is found.
2. If search reaches the last bucket without finding a matching item or empty-since-start bucket, the search continues at bucket 0.
3. If an empty-after-removal bucket is encountered, the algorithm continues to probe the next bucket.
4. If an empty-since-start bucket is encountered, the search algorithm returns null.



Consider the following hash table and a hash function of key % 10.

**valsTable:**

0	60	Empty-since-start
1		Empty-after-removal
2	110	Occupied
3		
4	364	
5	75	
6	66	
7		
8		
9	49	

- 1) HashSearch(valsTable, 75) probes \_\_\_\_\_ buckets.



/
**Check****Show answer**

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023

- 2) HashSearch(valsTable, 110) probes  
\_\_\_\_\_ buckets.

**Check****Show answer**

- 3) What does HashSearch(valsTable,  
112) return?

**Check****Show answer**

©zyBooks 11/20/23 11:08 104844  
Eric Quezada  
UTEP-CS2302-ValeraFall2023



- 4) HashSearch(valsTable, 207) probes  
\_\_\_\_\_ buckets.

**Check****Show answer**
**CHALLENGE ACTIVITY**

### 10.3.1: Linear probing.



502696.2096894.qx3zqy7

**Start**

valsTable:

0	50
1	61
2	
3	
4	
5	
6	46
7	
8	
9	

 Empty-since-start

 Empty-after-removal

 Occupied

Hash table valsTable uses linear probing and a hash function of key

HashInsert(valsTable, item 10) inserts item 10 into bucket  Ex: 10

HashInsert(valsTable, item 74) inserts item 74 into bucket

©zyBooks 11/20/23 11:08 104844  
Eric Quezada

HashInsert(valsTable, item 86) inserts item 86 into bucket

UTEP-CS2302-ValeraFall2023

1

2

3

4

**Check****Next**

# 10.4 Quadratic probing

## Overview and insertion

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023

A hash table with **quadratic probing** handles a collision by starting at the key's mapped bucket, and then quadratically searches subsequent buckets until an empty bucket is found. If an item's mapped bucket is H, the formula  $H + (c1 * i^2) + c2$  is used to determine the item's index in the hash table.  $c1$  and  $c2$  are programmer-defined constants for quadratic probing. Inserting a key uses the formula, starting with  $i = 0$ , to repeatedly search the hash table until an empty bucket is found. Each time an empty bucket is not found,  $i$  is incremented by 1. Iterating through sequential  $i$  values to obtain the desired table index is called the **probing sequence**.

### PARTICIPATION ACTIVITY

10.4.1: Hash table insertion using quadratic probing:  $c1 = 1$  and  $c2 = 1$ .



## Animation content:

undefined

## Animation captions:

1. When inserting 55, no collision occurs with the first computed index of 5. Inserting 66 also does not cause a collision.
2. Inserting 25 causes a collision with the first computed index of 5.
3.  $i$  is incremented to 1 and a new index of 7 is computed. Bucket 7 is empty and 25 is inserted.

Figure 10.4.1: HashInsert with quadratic probing.

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023

```
HashInsert(hashTable, item) {
    i = 0
    bucketsProbed = 0

    // Hash function determines initial bucket
    bucket = Hash(item->key) % N
    while (bucketsProbed < N) {
        // Insert item in next empty bucket
        if (hashTable[bucket] is Empty) {
            hashTable[bucket] = item
            return true
        }

        // Increment i and recompute bucket index
        // c1 and c2 are programmer-defined constants for quadratic
        probing
        i = i + 1
        bucket = (Hash(item->key) + c1 * i + c2 * i * i) % N

        // Increment number of buckets probed
        bucketsProbed = bucketsProbed + 1
    }
    return false
}
```

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302-ValeraFall2023

**PARTICIPATION  
ACTIVITY**

10.4.2: Insertion using quadratic probing.



Assume a hash function returns key % 16 and quadratic probing is used with  $c1 = 1$  and  $c2 = 1$ . Refer to the table below.

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302-ValeraFall2023

0	32
1	49
2	16
3	3
4	
5	99
6	64
7	23
8	
9	
10	42
11	11
12	
13	
14	
15	

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302-Valera-Fall2023

1) 32 was inserted before 16

- True
- False

2) Which value was inserted without collision?

- 99
- 64
- 23

3) What is the probing sequence when inserting 48 into the table?

- 8
- 0, 8
- 0, 2, 6, 12

4) How many bucket index computations were necessary to insert 64 into the table?

- 1
- 2
- 3

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302-Valera-Fall2023



- 5) If 21 is inserted into the hash table,  
what would be the insertion index?

- 5
- 9
- 11

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023

## Search and removal

The search algorithm uses the probing sequence until the key being searched for is found or an empty-since-start bucket is found. The removal algorithm searches for the key to remove and, if found, marks the bucket as empty-after-removal.

PARTICIPATION  
ACTIVITY

10.4.3: Search and removal with quadratic probing:  $c_1 = 1$  and  $c_2 = 1$ .



### Animation captions:

1. 16, 32, and 64 all have a mapped bucket of 0. 32 was inserted first, then 16, then 64.
2. A search for 64 iterates through indices in the probe sequence: 0, 2, then 6.
3. Removal of 32 marks bucket 0 as empty-after removal.
4. A search for 64 after removing item 32 checks the empty-after-removal bucket at index 0, searches the occupied bucket at index 2, and then finds item 64 at index 6.

Figure 10.4.2: HashRemove and HashSearch with quadratic probing.

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023

```
HashRemove(hashTable, key) {
    i = 0
    bucketsProbed = 0

    // Hash function determines initial bucket
    bucket = Hash(key) % N

    while ((hashTable[bucket] is not EmptySinceStart) and (bucketsProbed <
N)) {
        if ((hashTable[bucket] is Occupied) and (hashTable[bucket]→key == key)) {
            hashTable[bucket] = EmptyAfterRemoval
            return true
        }

        // Increment i and recompute bucket index
        // c1 and c2 are programmer-defined constants for quadratic probing
        i = i + 1
        bucket = (Hash(key) + c1 * i + c2 * i * i) % N

        // Increment number of buckets probed
        bucketsProbed = bucketsProbed + 1
    }
    return false // key not found
}

HashSearch(hashTable, key) {
    i = 0
    bucketsProbed = 0

    // Hash function determines initial bucket
    bucket = Hash(key) % N

    while ((hashTable[bucket] is not EmptySinceStart) and (bucketsProbed <
N)) {
        if ((hashTable[bucket] is Occupied) and (hashTable[bucket]→key == key)) {
            return hashTable[bucket]
        }

        // Increment i and recompute bucket index
        // c1 and c2 are programmer-defined constants for quadratic probing
        i = i + 1
        bucket = (Hash(key) + c1 * i + c2 * i * i) % N

        // Increment number of buckets probed
        bucketsProbed = bucketsProbed + 1
    }
    return null // key not found
}
```

©zyBooks 11/20/23 11:08 1048447

Eric Quezada  
UTEPCS2302ValeraFall2023

Consider the following hash table, a hash function of key % 10, and quadratic probing with  $c_1 = 1$  and  $c_2 = 1$ .

valsTable:	<table border="1"> <tbody> <tr><td>0</td><td>60</td></tr> <tr><td>1</td><td></td></tr> <tr><td>2</td><td>110</td></tr> <tr><td>3</td><td></td></tr> <tr><td>4</td><td>364</td></tr> <tr><td>5</td><td>75</td></tr> <tr><td>6</td><td>66</td></tr> <tr><td>7</td><td></td></tr> <tr><td>8</td><td></td></tr> <tr><td>9</td><td>49</td></tr> </tbody> </table>	0	60	1		2	110	3		4	364	5	75	6	66	7		8		9	49	<ul style="list-style-type: none"> <li><span style="color: lightblue;">□</span> Empty-since-start</li> <li><span style="color: lightgray;">□</span> Empty-after-removal</li> <li><span style="color: orange;">□</span> Occupied</li> </ul>
0	60																					
1																						
2	110																					
3																						
4	364																					
5	75																					
6	66																					
7																						
8																						
9	49																					

- 1) HashSearch(valsTable, 75) probes  
\_\_\_\_ buckets.

11

## Check

Show answer

- 2) HashSearch(valsTable, 110) probes \_\_\_\_\_ buckets.

11

### Check

### Show answer

- 3) After removing 66 via  
HashRemove(valsTable, 66),  
HashSearch(valsTable, 66) probes  
\_\_\_\_\_ buckets.

11

## Check

### Show answer

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302-Valera-Fall2023

## PARTICIPATION ACTIVITY

#### 10.4.5: Using empty buckets during search, insertion, and removal.



- 1) When a hash table is initialized, all entries must be empty-after-removal.

True  
 False

- 2) The insertion algorithm can only insert into empty-since-start buckets.

True  
 False

- 3) The search algorithm stops only when encountering a bucket containing the key being searched for.

True  
 False

- 4) The removal algorithm searches for the bucket containing the key to remove. If found, the bucket is marked as empty-after-removal.

True  
 False

**CHALLENGE ACTIVITY**

10.4.1: Quadratic probing.



502696.2096894.qx3zqy7

Start

valsTable:	0
	1 31
	2
	3
	4
	5
	6
	7
	8
	9 29

- Empty-since-start
- Empty-after-removal
- Occupied

Hash table valsTable uses quadratic probing, a hash function of key % 10, c1 = 1, and c2 = 1.

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEPACS2302ValeraFall2023

HashInsert(valsTable, item 39) inserts item 39 into bucket Ex: 10

HashInsert(valsTable, item 61) inserts item 61 into bucket

HashInsert(valsTable, item 40) inserts item 40 into bucket

1

2

3

4

[Check](#)[Next](#)

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023

## 10.5 Double hashing

### Overview

**Double hashing** is an open-addressing collision resolution technique that uses 2 different hash functions to compute bucket indices. Using hash functions  $h_1$  and  $h_2$ , a key's index in the table is computed with the formula  $i = (h_1(key) + i \cdot h_2(key)) \% \text{tableSize}$ . Inserting a key uses the formula, starting with  $i = 0$ , to repeatedly search hash table buckets until an empty bucket is found. Each time an empty bucket is not found,  $i$  is incremented by 1. Iterating through sequential  $i$  values to obtain the desired table index is called the **probing sequence**.

**PARTICIPATION ACTIVITY**

10.5.1: Hash table insertion using double hashing.



### Animation captions:

1. Items 72, 60, 45, 18, and 39 are inserted without collisions.
2. When inserting item 55, bucket 5 is occupied. Incrementing  $i$  to 1 and recomputing the hash function yields an empty bucket at index 3 for item 55.
3. Inserting item 23 also result in collisions.  $i$  is incremented to 2 before finding an empty bucket at index 10 to insert item 23.

**PARTICIPATION ACTIVITY**

10.5.2: Double hashing.



Given:

 $\text{hash1}(\text{key}) = \text{key \% } 11$  $\text{hash2}(\text{key}) = 5 - \text{key \% } 5$ 

and a hash table with a size of 11. Determine the index for each item after the following operations have been executed.

`HashInsert(valsTable, item 16)``HashInsert(valsTable, item 77)`

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023

```
HashInsert(valsTable, item 55)
HashInsert(valsTable, item 41)
HashInsert(valsTable, item 63)
```

1) Item 16

 **Check****Show answer**

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEPACS2302ValeraFall2023

2) Item 55

 **Check****Show answer**

3) Item 63

 **Check****Show answer**

## Insertion, search, and removal

Using double hashing, a hash table search algorithm probes (or checks) each bucket using the probing sequence defined by the two hash functions. The search continues until either the matching item is found (returning the item), an empty-since-start bucket is found (returning null), or all buckets are probed without a match (returning null).

A hash table insert algorithm probes each bucket using the probing sequence, and inserts the item in the next empty bucket (the empty kind doesn't matter).

A hash table removal algorithm first searches for the item's key. If the item is found, the item is removed, and the bucket is marked empty-after-removal.

**PARTICIPATION ACTIVITY**

10.5.3: Hash table insertion, search, and removal using double hashing.



©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEPACS2302ValeraFall2023

### Animation captions:

1. When item 3 is removed, the bucket is marked as empty-after-removal.
2. Search for 19 checks bucket 3 first. The bucket is empty-after-removal, and additional buckets must be searched.
3. Inserting item 88 has a collision at bucket 8. The next bucket index of 3 yields an empty-after-removal bucket, and item 88 is inserted in that bucket.

**PARTICIPATION ACTIVITY**

10.5.4: Hash table with double hashing: search, insert, and remove.



Consider the following hash table, a first hash function of key % 10, and a second hash function of 7 - key % 7.

valsTable:	0	60
	1	
	2	
	3	223
	4	104
	5	
	6	66
	7	
	8	
	9	49

- Empty-since-start
- Empty-after-removal
- Occupied

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302-ValeraFall2023

- 1) HashSearch(valsTable, 110) probes \_\_\_\_\_ buckets.

**Check****Show answer**

- 2) HashInsert(valsTable, item 24) probes \_\_\_\_\_ buckets.

**Check****Show answer**

- 3) After removing 66 via  
HashRemove(valsTable, 66),  
HashSearch(valsTable, 66) probes  
\_\_\_\_\_ buckets.

**Check****Show answer**

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302-ValeraFall2023

**PARTICIPATION ACTIVITY**

10.5.5: Hash table insertion, search, and removal with double hashing.





1) When the removal algorithm finds the bucket containing the key to be removed, the bucket is marked as empty-since-start.

- True
- False

2) Double hashing would never resolve collisions if the second hash function always returned 0.

- True
- False

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302-ValeraFall2023



### CHALLENGE ACTIVITY

#### 10.5.1: Double hashing.



502696.2096894.qx3zqy7

**Start**

valsTable:	0
	1
	2 79
	3
	4
	5 27
	6
	7
	8
	9
	10

- Empty-since-start
- Empty-after-removal
- Occupied

Hash table valsTable uses double probing with the hash functions  
 $\text{hash1(key)}: \text{key \% 11}$   
 $\text{hash2(key)}: 7 - \text{key \% 7}$   
and a table size of 11.

HashInsert(valsTable, item 20) inserts item 20 into bucket Ex: 10

HashInsert(valsTable, item 46) inserts item 46 into bucket

HashInsert(valsTable, item 24) inserts item 24 into bucket

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302-ValeraFall2023

1

2

3

**Check**

**Next**

# 10.6 Hash table resizing

## Resize operation

A hash table **resize** operation increases the number of buckets, while preserving all existing items. A hash table with  $N$  buckets is commonly resized to the next prime number  $\geq N * 2$ . A new array is allocated, and all items from the old array are re-inserted into the new array, making the resize operation's time complexity  $O(N)$ .

### PARTICIPATION ACTIVITY

10.6.1: Hash table resize operation.



## Animation content:

undefined

## Animation captions:

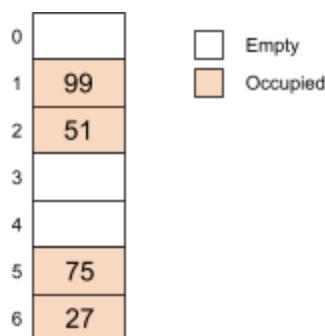
1. When resizing a hash table with 11 buckets and 7 items, the new size is computed as the next prime number  $\geq 22$ , which is 23.
2. A new array is allocated with 23 buckets for the resized hash table.
3. When rehashing 88, the bucket index is computed as  $88 \% 23 = 19$ . newArray[19] is assigned with 88.
4. The key from each of hashTable's non-empty buckets is rehashed and inserted into newArray.
5. newArray is returned and is the resized hash table.

### PARTICIPATION ACTIVITY

10.6.2: Resizing a hash table.



Suppose the hash table below is resized. The hash function used both before and after resizing is:  $\text{hash(key)} = \text{key \% } N$ , where  $N$  is the table size.



©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEPACS2302ValeraFall2023



- 1) What is the most likely allocated size for the resized hash table?

- 7
- 14
- 17

- 2) How many elements are in the hash table after resizing?

- 0
- 4
- 7

- 3) At what index does 99 reside in the resized table?

- 1
- 9
- 14

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEPACS2302ValeraFall2023

## When to resize

A hash table's **load factor** is the number of items in the hash table divided by the number of buckets.  
Ex: A hash table with 18 items and 31 buckets has a load factor of  $\frac{18}{31}$ . The load factor may be used to decide when to resize the hash table.

An implementation may choose to resize the hash table when one or more of the following values exceeds a certain threshold:

- Load factor
- When using open-addressing, the number of collisions during an insertion
- When using chaining, the size of a bucket's linked-list

PARTICIPATION ACTIVITY

10.6.3: Resizing when a chaining bucket is too large.



©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEPACS2302ValeraFall2023

## Animation captions:

1. A hash table with chaining will inevitably have large linked-lists after inserting many items.
2. The largest bucket length can be used as resizing criteria. Ex: The hash table is resized when a bucket length is  $\geq 4$ .

**PARTICIPATION ACTIVITY**10.6.4: Resizing when the load factor is  $\geq 0.6$ .**Animation captions:**

1. A hash table with 2 items and 5 buckets has a load factor of  $2 / 5 = 0.4$ .
2. Inserting 22 increases the load factor to  $3 / 5 = 0.6$ .
3. An implementation may choose to resize the hash table whenever the load factor is  $\geq 0.6$ .

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEP-CS2302-Valera-Fall2023

**PARTICIPATION ACTIVITY**10.6.5: Resizing when an insertion causes more than  $N / 3$  collisions.**Animation captions:**

1. Inserting 38 into the hash table with linear probing encounters 5 collisions before placing 38 in bucket 10.
2. If the hash table's resize criteria were to resize after encountering  $N / 3$  collisions, then the insertion would cause a resize.

**PARTICIPATION ACTIVITY**

10.6.6: Resize criteria and load factors.



- 1) A hash table implementation must use only one criteria for resizing.

- True
- False



- 2) In a hash table using open addressing, the load factor cannot exceed 1.0.

- True
- False



- 3) In a hash table using chaining, the load factor cannot exceed 1.0.

- True
- False

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302-Valera-Fall2023

- 4) When resizing to a larger size, the load factor is guaranteed to decrease.

- True
- False

**PARTICIPATION  
ACTIVITY**

10.6.7: Resizing a hash table with 101 buckets.



Suppose a hash table has 101 buckets.

- 1) If the hash table was using chaining,  
the load factor could be  $\leq 0.1$ , but an  
individual bucket could still contain 10  
items.

- True  
 False

- 2) If the hash table was using open  
addressing, a load factor  $< 0.25$   
guarantees that no more than 25  
collisions will occur during insertion.

- True  
 False

- 3) If the hash table was using open  
addressing, a load factor  $> 0.9$   
guarantees a collision during insertion.

- True  
 False

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEPACS2302ValeraFall2023

## 10.7 Common hash functions

### A good hash function minimizes collisions

A hash table is fast if the hash function minimizes collisions.

A **perfect hash function** maps items to buckets with no collisions. A perfect hash function can be created if the number of items and all possible item keys are known beforehand. The runtime for insert, search, and remove is  $O(1)$  with a perfect hash function.

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEPACS2302ValeraFall2023

A good hash function should uniformly distribute items into buckets. With chaining, a good hash function results in short bucket lists and thus fast inserts, searches, and removes. With linear probing, a good hash function will avoid hashing multiple items to consecutive buckets and thus minimize the average linear probing length to achieve fast inserts, searches, and removes. On average, a good hash function will achieve  $O(1)$  inserts, searches, and removes, but in the worst-case may require  $O(N)$ .

A hash function's performance depends on the hash table size and knowledge of the expected keys.  
Ex: The hash function key % 10 will perform poorly if the expected keys are all multiples of 10, because inserting 10, 20, 30, ..., 90, and 100 will all collide at bucket 0.

## Modulo hash function

A **modulo hash** uses the remainder from division of the key by hash table size N.

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023

Figure 10.7.1: Modulo hash function.

```
HashRemainder(int key)
{
    return key % N
}
```

### PARTICIPATION ACTIVITY

10.7.1: Good hash functions and keys.



Will the hash function and expected key likely work well for the following scenarios?

1) Hash function: key % 1000



Key: 6-digit employee ID

Hash table size: 20000

- Yes
- No

2) Hash function: key % 250



Key: 5-digit customer ID

Hash table size: 250

- Yes
- No

3) Hash function: key % 1000



Key: Selling price of a house.

Hash table size: 1000

- Yes
- No

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023



4) Hash function: key % 40

Key: 4-digit even numbers

Hash table size: 40

- Yes
- No

5) Hash function: key % 1000

Key: Customer's 3-digit U.S. phone number area code, of which about 300 exist.

Hash table size: 1000

- Yes
- No

©zyBooks 11/20/23 11:08 1048417

Eric Quezada

UTEPACS2302ValeraFall2023

## Mid-square hash function

A **mid-square hash** squares the key, extracts R digits from the result's middle, and returns the remainder of the middle digits divided by hash table size N. Ex: For a hash table with 100 entries and a key of 453, the decimal (base 10) mid-square hash function computes  $453 * 453 = 205209$ , and returns the middle two digits 52. For N buckets, R must be greater than or equal to \_\_\_\_\_ to index all buckets. The process of squaring and extracting middle digits reduces the likelihood of keys mapping to just a few buckets.

### PARTICIPATION ACTIVITY

10.7.2: Decimal mid-square hash function.



1) For a decimal mid-square hash function, what are the middle digits for key = 40, N = 100, and R = 2?

**Check**

**Show answer**

2) For a decimal mid-square hash function, what is the bucket index for key = 110, N = 200, and R = 3?

**Check**

**Show answer**

©zyBooks 11/20/23 11:08 1048417

Eric Quezada

UTEPACS2302ValeraFall2023



- 3) For a decimal mid-square hash function, what is the bucket index for key = 112, N = 1000, and R = 3?

 //[Show answer](#)

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023

## Mid-square hash function base 2 implementation

The mid-square hash function is typically implemented using binary (base 2), and not decimal, because a binary implementation is faster. A decimal implementation requires converting the square of the key to a string, extracting a substring for the middle digits, and converting that substring to an integer. A binary implementation only requires a few shift and bitwise AND operations.

A binary mid-square hash function extracts the middle R bits, and returns the remainder of the middle bits divided by hash table size N, where R is greater than or equal to . Ex: For a hash table size of 200, R = 8, then 8 bits are needed for indices 0 to 199.

Figure 10.7.2: Mid-square hash function (base 2).

```
HashMidSquare(int key) {
    squaredKey = key * key

    lowBitsToRemove = (32 - R) / 2
    extractedBits = squaredKey >> lowBitsToRemove
    extractedBits = extractedBits & (0xFFFFFFFF >> (32 - R))

    return extractedBits % N
}
```

The extracted middle bits depend on the maximum key. Ex: A key with a value of 4000 requires 12 bits. A 12-bit number squared requires up to 24 bits. For R = 10, the middle 10 bits of the 24-bit squared key are bits 7 to 16.

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023



**PARTICIPATION  
ACTIVITY**

10.7.3: Binary mid-square hash function.



- 1) For a binary mid-square hash function, how many bits are needed for an 80 entry hash table?

 //**Check****Show answer**

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023

- 2) For R = 3, what are the middle bits for a key of 9?  $9 * 9 = 81$ ; 81 in binary is 1010001.

 //**Check****Show answer**

## Multiplicative string hash function

A **multiplicative string hash** repeatedly multiplies the hash value and adds the ASCII (or Unicode) value of each character in the string. A multiplicative hash function for strings starts with a large initial value. For each character, the hash function multiplies the current hash value by a multiplier (often prime) and adds the character's value. Finally, the function returns the remainder of the sum divided by the hash table size N.

Figure 10.7.3: Multiplicative string hash function.

```
HashMultiplicative(string key) {
    stringHash = initialValue

    for (each character strChar in key) {
        stringHash = (stringHash * HashMultiplier) +
    strChar
    }

    return stringHash % N
}
```

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023

Daniel J. Bernstein created a popular version of a multiplicative string hash function that uses an initial value of 5381 and a multiplier of 33. Bernstein's hash function performs well for hashing short English strings.

### PARTICIPATION ACTIVITY

10.7.4: Multiplicative string hash function.



For a 1000-entry hash table, compute the multiplicative hash for the following strings using the specific initial value and hash multiplier. The decimal ASCII value for each character is shown below.

Character	Decimal value	Character	Decimal value
A	65	N	78
B	66	O	79
C	67	P	80
D	68	Q	81
E	69	R	82
F	70	S	83
G	71	T	84
H	72	U	85
I	73	V	86
J	74	W	87
K	75	X	88
L	76	Y	89
M	77	Z	90

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023

- 1) Initial value = 0

Hash multiplier = 1

String = BAT



**Check**

**Show answer**



- 2) Initial value = 0

Hash multiplier = 1

String = TAB



**Check**

**Show answer**

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023





3) Initial value = 17

Hash multiplier = 3

String = WE

//

**Check**

**Show answer**

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023

Exploring further:

The following provide resources that summarize, discuss, and analyze numerous hash functions.

- [Hash Functions: An Empirical Comparison](#) by Peter Kankowski.
- [Hash Functions and Block Ciphers](#) by Bob Jenkins.

## 10.8 Direct hashing

### Direct hashing overview

A **direct hash function** uses the item's key as the bucket index. Ex: If the key is 937, the index is 937. A hash table with a direct hash function is called a **direct access table**. Given a key, a direct access table **search** algorithm returns the item at index key if the bucket is not empty, and returns null (indicating item not found) if empty.

PARTICIPATION  
ACTIVITY

10.8.1: Direct hash function.



#### Animation content:

undefined

#### Animation captions:

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023

1. A direct hash function uses the item's key as the bucket index. The value stored in hashTable[6] is returned.
2. hashTable[3] is empty. Search returns null, indicating the item was not found.

Figure 10.8.1: Direct hashing: Insert, remove, and search operations use item's key as bucket index.

```
HashInsert(hashTable, item) {  
    hashTable[item->key] = item  
}  
  
HashRemove(hashTable, item) {  
    hashTable[item->key] = Empty  
}  
  
HashSearch(hashTable, key) {  
    if (hashTable[key] is not Empty)  
    {  
        return hashTable[key]  
    }  
    else {  
        return null  
    }  
}
```

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302-Valera-Fall2023

PARTICIPATION  
ACTIVITY

10.8.2: Direct access table search, insert, and remove.



Type the hash table after the given operations. Type the hash table as: E, 1, 2, E, E (where E means empty).

1)

numsTable:	0
	1
	2



HashInsert(numsTable, item 0)

numsTable:

**Check**

**Show answer**

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302-Valera-Fall2023



2)

0	0
1	
2	2
3	3
4	

HashRemove(numsTable, 0)  
 HashInsert(numsTable, item 4)

numsTable:

 //

©zyBooks 11/20/23 11:08 1048447  
 Eric Quezada  
 UTEPCS2302ValeraFall2023

**Check****Show answer**

## Limitations of direct hashing

A direct access table has the advantage of no collisions: Each key is unique (by definition of a key), and each gets a unique bucket, so no collisions can occur. However, a direct access table has two main limitations.

1. All keys must be non-negative integers, but for some applications keys may be negative.
2. The hash table's size equals the largest key value plus 1, which may be very large.

**PARTICIPATION ACTIVITY**

10.8.3: Direct hashing limitations.



- 1) For a 1000-entry direct access table, type the bucket number for the inserted item, or type: None

HashInsert(hashIndex, item 734)

 //**Check****Show answer**

- 2) For a 1000-entry direct access table, type the bucket number for the inserted item, or type: None

HashInsert(hashIndex, item 1034)

 //**Check****Show answer**

©zyBooks 11/20/23 11:08 1048447  
 Eric Quezada  
 UTEPCS2302ValeraFall2023



- 3) For a 1000-entry direct access table, type the bucket number for the inserted item, or type: None

HashInsert(hashIndex, item -45)

 [Show answer](#)

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302-ValeraFall2023

- 4) How many direct access table buckets are needed for items with keys ranging from 100 to 200 (inclusive)?

 [Check](#)[Show answer](#)

- 5) A class has 100 students. Student ID numbers range from 10000 to 99999. Using the ID number as key, how many buckets will a direct access table require?

 [Check](#)[Show answer](#)

## 10.9 Hashing Algorithms: Cryptography, Password Hashing

### Cryptography

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302-ValeraFall2023

**Cryptography** is a field of study focused on transmitting data securely. Secure data transmission commonly starts with **encryption**: alteration of data to hide the original meaning. The counterpart to encryption is **decryption**: reconstruction of original data from encrypted data.

#### PARTICIPATION ACTIVITY

10.9.1: Basic encryption: Caesar cipher.



## Animation captions:

1. The Caesar cipher shifts characters in the alphabet to encrypt a message. With a right shift of 4, the character 'N' is shifted to 'R'.
2. The shift is applied to each character in the string, including spaces. The result is an encrypted message that hides the original message.
3. A left shift can also be used.
4. Each message can be decrypted with the opposite shift.

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEPACS2302ValeraFall2023

### PARTICIPATION ACTIVITY

10.9.2: Caesar cipher.



- 1) What is the result of applying the Caesar cipher with a left shift of 1 to the string "computer"?
  - eqorwvgt
  - dpnqvufs
  - bnlotdsdq
- 2) If a message is encrypted with a left shift of X, what shift is needed to decrypt?
  - left shift of X
  - right shift of X
- 3) If the Caesar cipher were implemented such that strings were restricted to only lower-case alphabet characters, how many distinct ways could a message be encrypted?
  - 25
  - 50
  - Length of the message



©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEPACS2302ValeraFall2023

### PARTICIPATION ACTIVITY

10.9.3: Cryptography.





1) Encryption and decryption are synonymous.

- True
- False

2) Cryptography is used heavily in internet communications.

- True
- False

3) The Caesar cipher is an encryption algorithm that works well to secure data for modern digital communications.

- True
- False

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEP-CS2302-Valera-Fall2023

## Hashing functions for data

A hash function can be used to produce a hash value for data in contexts other than inserting the data into a hash table. Such a function is commonly used for the purpose of verifying data integrity. Ex: A hashing algorithm called MD5 produces a 128-bit hash value for any input data. The hash value cannot be used to reconstruct the original data, but can be used to help verify that data isn't corrupt and hasn't been altered.

PARTICIPATION ACTIVITY

10.9.4: A hash value can help identify corrupted data downloaded from the internet.



### Animation captions:

1. Computer B will attempt to download a message over the internet from computer A. Computer B will also download a corresponding 128-bit MD5 hash value from computer A.
2. Due to an unreliable network, the message data arrives corrupted. The MD5 hash is downloaded correctly.
3. Computer B computes the MD5 hash for the downloaded data. The computed hash is different from the downloaded hash, implying the data was corrupted.

©zyBooks 11/20/23 11:08 1048447

Eric Quezada  
UTEP-CS2302-Valera-Fall2023

PARTICIPATION ACTIVITY

10.9.5: Hashing functions for data.





- 1) MD5 produces larger hash values for larger input data sizes.
- True
- False

- 2) A hash value can be used to reconstruct the original data.
- True
- False

- 3) If computer B in the above example computed a hash value identical to the downloaded hash value, then the downloaded message would be guaranteed to be uncorrupted.
- True
- False

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEPACS2302ValeraFall2023



## Cryptographic hashing

A **cryptographic hash function** is a hash function designed specifically for cryptography. Such a function is commonly used for encrypting and decrypting data.

A **password hashing function** is a cryptographic hashing function that produces a hash value for a password. Databases for online services commonly store a user's password hash as opposed to the actual password. When the user attempts a login, the supplied password is hashed, and the hash is compared against the database's hash value. Because the passwords are not stored, if a database with password hashes is breached, attackers may still have a difficult time determining a user's password.

**PARTICIPATION ACTIVITY**

10.9.6: Password hashing function.



### Animation captions:

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEPACS2302ValeraFall2023

1. A password hashing function produces a hash value for a password.
2. The password hash function aims to produce a hash that cannot easily be converted back to the password.
3. Also, two different passwords should not produce the same hash value.
4. Some password hashing functions concatenate extra random data to the password, then store the random data as well as the password hash value.



1) Which is not an advantage of storing password hash values, instead of actual passwords, in a database?

- Database administrators cannot see users' passwords.
- Database storage space is saved.
- Attackers who gain access to database contents still may not be able to determine users' passwords.

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023



2) A user could login with an incorrect password if a password hashing function produced the same hash value for two different passwords.

- True
- False



3) Generating and storing random data alongside each password hash in a database, and using (password + random\_data) to generate the hash value, can help increase security.

- True
- False



## 10.10 Python: Hash tables

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEPACS2302ValeraFall2023

### HashTable base class

HashTable is a base class that declares methods for hashKey(), insert(), remove(), and search(). The hashKey() method returns a non-negative integer hash code for a key. Each of the insert(), remove(), and search() methods has only a pass statement and will be overridden in a derived class.

HashTable's hashKey() method uses Python's built-in hash() function. hash() may return a negative integer, but a non-negative integer is needed to compute a bucket index. So hashKey() returns the

absolute value of the integer returned from hash().

Figure 10.10.1: HashTable base class.

```
# Base class for a hash table that supports the insert, remove, and search
# operations.
class HashTable:
    # Returns a non-negative hash code for the specified key.
    def hashKey(self, key):
        return abs(hash(key))

    # Inserts the specified key/value pair. If the key already exists, the
    # corresponding value is updated. If inserted or updated, True is
    # returned.
    # If not inserted, then False is returned.
    def insert(self, key, value):
        pass

    # Searches for the specified key. If found, the key/value pair is
    # removed
    # from the hash table and True is returned. If not found, False is
    # returned.
    def remove(self, key):
        pass

    # Searches for the key, returning the corresponding value if found,
    # None if
    # not found.
    def search(self, key):
        pass
```

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302ValeraFall2023

PARTICIPATION  
ACTIVITY

10.10.1: HashTable base class.



1) Classes that derive from HashTable



must override hashKey().

True

False

2) Classes that derive from HashTable



should override insert(), remove(), and  
search().

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302ValeraFall2023

True

False



3) hashKey() may return a negative integer.

- True
- False

## Chaining

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEP-CS2302ValeraFall2023

The ChainingHashTableItem class represents an item with a key, a value, and a reference to the next item in the chain. ChainingHashTable derives from the HashTable class, and uses a list of ChainingHashTableItem objects for the **table** attribute. Initially each item in the table list is None, indicating an empty bucket.

ChainingHashTable's insert() method computes the key's bucket index, then searches the bucket's linked list for a matching key. If found, the corresponding value is updated. If not found, a new ChainingHashTableItem is appended to the linked list.

ChainingHashTable's remove() method computes the key's bucket index, then searches the bucket's linked list for a matching key. If found, the ChainingHashTableItem object with the key is removed from the bucket's linked list.

ChainingHashTable's search() method computes the key's bucket index, then searches the bucket's linked list for a matching key. If found, the key's corresponding value is returned. If not found, None is returned.

Figure 10.10.2: ChainingHashTableItem and ChainingHashTable classes.

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEP-CS2302ValeraFall2023

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302-Valera-Fall2023

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302-Valera-Fall2023

```
class ChainingHashTableItem:
    def __init__(self, itemKey, itemValue):
        self.key = itemKey
        self.value = itemValue
        self.next = None

class ChainingHashTable(HashTable):
    def __init__(self, initialCapacity = 11):
        self.table = [None] * initialCapacity

    # Inserts the specified key/value pair. If the key already exists, the
    # corresponding value is updated. If inserted or updated, True is
    # returned.
    # If not inserted, then False is returned.
    def insert(self, key, value):
        # Hash the key to get the bucket index
        bucket_index = self.hashKey(key) % len(self.table)

        # Traverse the linked list, searching for the key. If the key
        exists in
            # an item, the value is replaced. Otherwise a new item is appended.
        item = self.table[bucket_index]
        previous = None
        while item != None:
            if key == item.key:
                item.value = value
                return True

            previous = item
            item = item.next

        # Append to the linked list
        if self.table[bucket_index] == None:
            self.table[bucket_index] = ChainingHashTableItem(key, value)
        else:
            previous.next = ChainingHashTableItem(key, value)
        return True

    # Searches for the specified key. If found, the key/value pair is
    removed
    # from the hash table and True is returned. If not found, False is
    returned.
    def remove(self, key):
        # Hash the key to get the bucket index
        bucket_index = self.hashKey(key) % len(self.table)

        # Search the bucket's linked list for the key
        item = self.table[bucket_index]
        previous = None
        while item != None:
            if key == item.key:
                if previous == None:
                    # Remove linked list's first item
                    self.table[bucket_index] = item.next
                else:
                    previous.next = item.next
                return True
            previous = item
            item = item.next
```

©zyBooks 11/20/23 11:08 1048447

Eric Quezada

UTEP-CS2302-ValeraFall2023

```

        return False # key not found

    # Searches for the key, returning the corresponding value if found,
    None if
    # not found.
    def search(self, key):
        # Hash the key to get the bucket index
        bucket_index = self.hashKey(key) % len(self.table)

        # Search the bucket's linked list for the key
        item = self.table[bucket_index]
        while item != None:
            if key == item.key:
                return item.value
            item = item.next
        return None # key not found

```

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302-ValeraFall2023

## insert() method variations

Some hash table implementations distinguish an insert operation from an update operation. In such implementations, insert fails if the key already exists and update fails if the key doesn't already exist. The implementation of insert() in this section serves both purposes, inserting if the key doesn't exist and updating if the key does exist.

### PARTICIPATION ACTIVITY

10.10.2: ChainingHashTable class.



If unable to drag and drop, refresh the page.

**search()**

**insert()**

**remove()**

Returns False and makes no change to a bucket's linked list if the key is not found.

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302-ValeraFall2023

Returns None if the key is not found.

May return True without changing the length of a bucket's linked list.

**Reset**

## OpenAddressingBucket class

Each bucket in an open addressing hash table stores a single item, rather than a linked list of items. So the OpenAddressingBucket class has only two members: key and value.

Two empty bucket types exist in open addressing hash tables, so the OpenAddressingBucket class also defines two class attributes. Each attribute is an OpenAddressingBucket object, used to mark a bucket as empty-since-start or empty-after-removal.

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302-Valera-Fall2023

Figure 10.10.3: OpenAddressingBucket class.

```
# OpenAddressingBucket represents a bucket with a key and a value
class OpenAddressingBucket:
    def __init__(self, bucket_key = None, bucket_value = None):
        self.key = bucket_key
        self.value = bucket_value

    def isEmpty(self):
        if self is OpenAddressingBucket.EMPTY_SINCE_START:
            return True
        return self is OpenAddressingBucket.EMPTY_AFTER_REMOVAL

# Initialize two special bucket types: empty-since-start and empty-after-
# removal
OpenAddressingBucket.EMPTY_SINCE_START = OpenAddressingBucket()
OpenAddressingBucket.EMPTY_AFTER_REMOVAL = OpenAddressingBucket()
```

PARTICIPATION ACTIVITY

10.10.3: OpenAddressingBucket class.



- 1) Only one instance of EMPTY\_SINCE\_START exists, no matter how many OpenAddressingBucket instances are created.

- True
- False



- 2) An open addressing hash table implementation that has a list of OpenAddressingBucket references should initialize each to None.

- True
- False

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302-Valera-Fall2023





3) To check if a bucket is empty-after-removal, the bucket's key and value attributes must be compared with EMPTY\_AFTER\_REMOVAL's key and value attributes.

- True
- False

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302-Valera-Fall2023

## OpenAddressingHashTable base class

OpenAddressingHashTable is a base class that derives from HashTable and adds the probe() method. Each derived class implements the probing sequence via the probe() method, allowing for the insert(), remove(), and search() methods to be implemented in the OpenAddressingHashTable class.

The insert() method first uses the probing sequence to search for a bucket with the key. If found, the corresponding value is updated. If not found, the probing sequence is used again to find the first empty bucket, wherein the key/value pair is inserted.

The remove() method uses the probing sequence to search for a bucket with the key. If found, the bucket is set to EMPTY\_AFTER\_REMOVAL and True is returned. If not found, no change occurs and False is returned.

The search() method uses the probing sequence to search for a bucket with the key. If found, the bucket's value is returned. If not found, None is returned.

Figure 10.10.4: OpenAddressingHashTable class.

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302-Valera-Fall2023

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEPACS2302ValeraFall2023

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEPACS2302ValeraFall2023

```
class OpenAddressingHashTable(HashTable):
    def __init__(self, initialCapacity):
        self.table = [OpenAddressingBucket.EMPTY_SINCE_START] * initialCapacity

    def probe(self, key, i):
        # Each derived class must implement
        pass

        # Inserts the specified key/value pair. If the key already exists, the
        # corresponding value is updated. If inserted or updated, True is
        returned.
        # If not inserted, then False is returned.
    def insert(self, key, value):
        # Search for the key in the table. If found, update the bucket's
        value.
        for i in range(len(self.table)):
            bucket_index = self.probe(key, i)

            # An empty-since-start bucket implies the key is not in the
            table
            if self.table[bucket_index] is
OpenAddressingBucket.EMPTY_SINCE_START:
                break

            if self.table[bucket_index] is not
OpenAddressingBucket.EMPTY_AFTER_REMOVAL:
                # Check if the non-empty bucket has the key
                if key == self.table[bucket_index].key:
                    # Update the value
                    self.table[bucket_index].value = value
                    return True

            # The key is not in the table, so insert into first empty bucket
            for i in range(len(self.table)):
                bucket_index = self.probe(key, i)
                if self.table[bucket_index].isEmpty():
                    self.table[bucket_index] = OpenAddressingBucket(key, value)
                    return True

        return False # no empty bucket found

        # Searches for the specified key. If found, the key/value pair is
        removed
        # from the hash table and True is returned. If not found, False is
        returned.
    def remove(self, key):
        for i in range(len(self.table)):
            bucket_index = self.probe(key, i)

            # An empty-since-start bucket implies the key is not in the
            table
            if self.table[bucket_index] is
OpenAddressingBucket.EMPTY_SINCE_START:
                return False

            if self.table[bucket_index] is not
OpenAddressingBucket.EMPTY_AFTER_REMOVAL:
                # Check if the non-empty bucket has the key
```

```

        if key == self.table[bucket_index].key:
            # Remove by setting the bucket to empty-after-removal
            self.table[bucket_index] =
OpenAddressingBucket.EMPTY_AFTER_REMOVAL
            return True
        return False # key not found

    # Searches for the key, returning the corresponding value if found,
    None if
        # not found.
    def search(self, key):
        for i in range(len(self.table)):
            bucket_index = self.probe(key, i)

            # An empty-since-start bucket implies the key is not in the
            table
            if self.table[bucket_index] is
OpenAddressingBucket.EMPTY_SINCE_START:
                return None

            if self.table[bucket_index] is not
OpenAddressingBucket.EMPTY_AFTER_REMOVAL:
                # Check if the non-empty bucket has the key
                if key == self.table[bucket_index].key:
                    return self.table[bucket_index].value

    return None # key not found

```

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302-ValeraFall2023

#### PARTICIPATION ACTIVITY

##### 10.10.4: OpenAddressingHashTable class.



- 1) Each class that derives from the OpenAddressingHashTable class must implement the probe(), insert(), remove(), and search() methods.

- True
- False



- 2) The OpenAddressingHashTable class's \_\_init\_\_() method initializes all pointers in the **table** list to:
- OpenAddressingBucket.EMPTY\_SINCE\_START**

- True
- False

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302-ValeraFall2023



- 3) The insert(), remove(), and search() methods each call the probe() method.
- True
  - False



## Linear probing, quadratic probing, and double hashing

The LinearProbingHashTable, QuadraticProbingHashTable, and DoubleHashingHashTable classes each derive from OpenAddressingHashTable. Since insert(), remove(), and search() are implemented in OpenAddressingHashTable, each derived class only implements the probing sequence.

Figure 10.10.5: LinearProbingHashTable class.

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302-ValeraFall2023

```
class LinearProbingHashTable(OpenAddressingHashTable):
    def __init__(self, initial_capacity = 11):
        OpenAddressingHashTable.__init__(self, initial_capacity)

    # Returns the bucket index for the specified key and i value using
    # the
    # linear probing sequence.
    def probe(self, key, i):
        return (self.hashKey(key) + i) % len(self.table)
```

Figure 10.10.6: QuadraticProbingHashTable class.

```
class QuadraticProbingHashTable(OpenAddressingHashTable):
    def __init__(self, c1 = 1, c2 = 1, initial_capacity = 11):
        OpenAddressingHashTable.__init__(self, initial_capacity)
        self.c1 = c1
        self.c2 = c2

    # Returns the bucket index for the specified key and i value using the
    # quadratic probing sequence.
    def probe(self, key, i):
        return (self.hashKey(key) + self.c1 * i + self.c2 * i * i) %
len(self.table)
```

Figure 10.10.7: DoubleHashingHashTable class.

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302-ValeraFall2023

```
class DoubleHashingHashTable(OpenAddressingHashTable):
    def __init__(self, initial_capacity = 11):
        OpenAddressingHashTable.__init__(self, initial_capacity)

    def h2(self, key):
        return 7 - self.hashKey(key) % 7

    # Returns the bucket index for the specified key and i value using
    # the
    # double hashing probing sequence.
    def probe(self, key, i):
        return (self.hashKey(key) + i * self.h2(key)) % len(self.table)
```

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302 Valera Fall 2023

**PARTICIPATION ACTIVITY**

10.10.5: Linear probing, quadratic probing, and double hashing.



1) Which class implements an additional

hash function?



- LinearProbingHashTable
- QuadraticProbingHashTable
- DoubleHashingHashTable

2) When 0 is passed as the second

argument to the probe() method, each  
class's returned value equals \_\_\_\_.



- hash(key)
- self.hashKey(key) %  
len(self.table)
- c1 + c2

**PARTICIPATION ACTIVITY**

10.10.6: Evaluating the probe() method.



Assume for each question that the same key object is used, `self.hashKey(key)` returns

55, and the table's length is 11.

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEP-CS2302 Valera Fall 2023

1) `probe(key, 2)` returns \_\_\_\_ for  
LinearProbingHashTable.



- 2
- 7
- 57



2) `probe(key, 1)` returns \_\_\_ for QuadraticProbingHashTable when  $c1=2$  and  $c2=2$ .

- 2
- 4
- 6

3) `probe(key, 4)` returns \_\_\_ for DoubleHashingHashTable.

- 4
- 5
- 6

©zyBooks 11/20/23 11:08 1048447

Eric Quezada  
UTEPACS2302ValeraFall2023

zyDE 10.10.1: Working with the Hashtable class.

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEPACS2302ValeraFall2023

The program below shows the bucket contents after adding and removing items to each of hash table. The `__str__()` method is implemented in ChainingHashTable and OpenAddressingHashTable to create a visual representation of the buckets.

Airport locations are used as keys and airport codes as corresponding values.

Current **HashTablesDemo.py** ▾ [Load default template](#)

file: [HashTablesDemo.py](#) | [Edit](#) | [History](#) | [Raw](#) | [Download](#) | [Run](#) | [Copy](#) | [Delete](#)

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEPCS2302ValeraFall2023

```
1 from ChainingHashTable import ChainingHashTable
2 from LinearProbingHashTable import LinearProbingHashTable
3 from QuadraticProbingHashTable import QuadraticProbingHashTable
4 from DoubleHashingHashTable import DoubleHashingHashTable
5
6 keys = [
7     "Los Angeles", "Houston", "Washington",
8     "Chicago", "San Francisco", "Dallas",
9     "Tokyo", "New York", "Vancouver"
10 ]
11 values = [
12     "LAX", "IAH", "IAD",
13     "ORD", "SFO", "DAL",
14     "NRT", "JFK", "YVR"
15 ]
16 :
```

**Run**

## 10.11 LAB: Course gradebook with dict



This section's content is not available for print.

©zyBooks 11/20/23 11:08 1048447  
Eric Quezada  
UTEPCS2302ValeraFall2023