13.1 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 13.1.1: 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 13.1.2: 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

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?

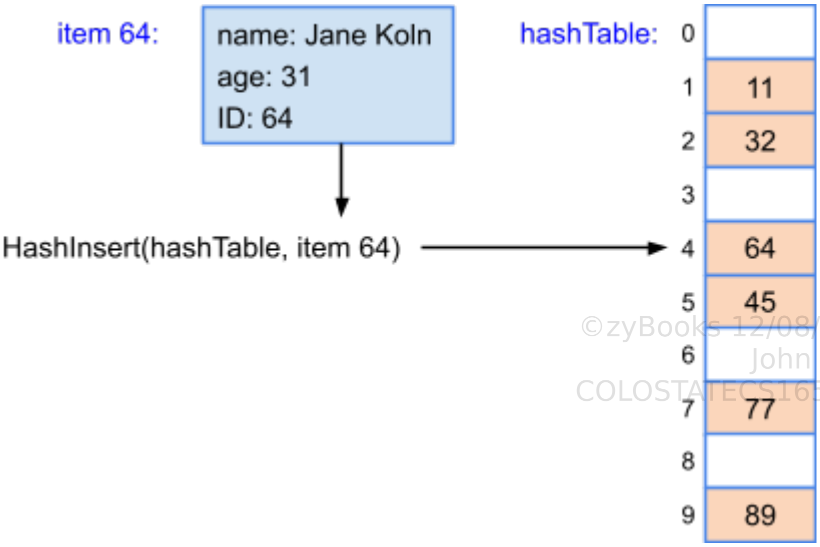
○ Name

○ Department

○ Employee ID number

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 | 13.1.3: Hash tables.

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?

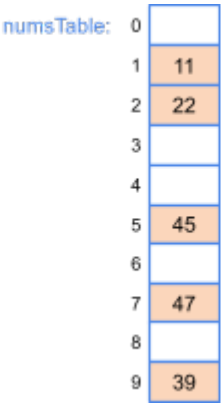
Check Show answer

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

PARTICIPATION ACTIVITY | 13.1.4: Hash table search efficiency.

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



numsTable:

0	
1	11
2	22
3	
4	
5	45
6	
7	47
8	
9	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

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

○ 1

○ 6

○ 10

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 | 13.1.5: 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

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)

○ Item 55

○ Item 90

○ Item 95

CHALLENGE ACTIVITY | 13.1.1: Hash tables with modulo hash function.

422352.2723990.qx3zqy7

Start

A hash table with non-negative integer keys has a modulo hash function of key % 25.

Hash function index range: 0 to [Ex: 5]

Item 177 will go in bucket [Ex: 26]

1 2 3

Check Next

13.2 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 | 13.2.1: Hash table with chaining.

Animation content:

undefined

Animation captions:

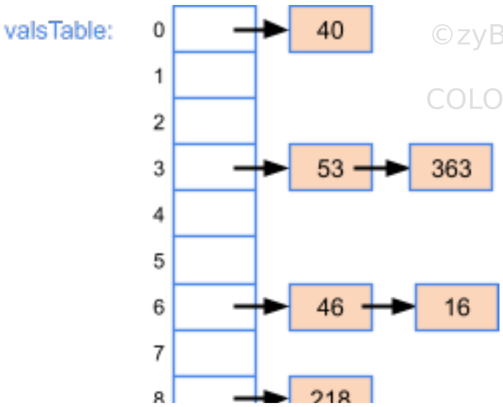
1. A hash table with chaining uses a list for each bucket. The insert operation first uses the item's key to determine the mapped bucket, and then inserts the item in that bucket's list.
2. A bucket may store multiple items with different keys that map to the same bucket. If collisions occur, items are 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 13.2.1: Hash table with chaining: Each bucket contains a list of items.

```
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
}
```

PARTICIPATION ACTIVITY 13.2.2: 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).



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

3) HashRemove(valsTable, 46)
Bucket 6's list: _____

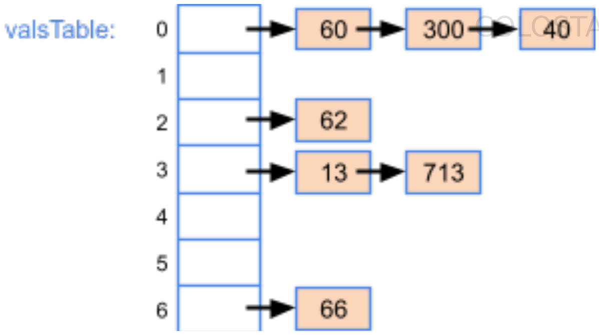
Check Show answer

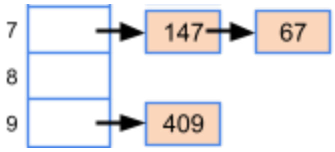
4) HashRemove(valsTable, 218)
Bucket 8's list: _____

Check Show answer

PARTICIPATION ACTIVITY 13.2.3: Hash table with chaining: Search.

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





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

Check Show answer

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 13.2.1: Chaining.

422352.2723990.qx3zqy7

Start

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

1	2	3	4	5

Check Next

13.3 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 13.3.1: Hash table with linear probing.

Animation captions:

1. 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.
2. The item is inserted in the next empty bucket.

3. Search starts at the hashed location and will compare each bucket until a match is found.
4. If an empty bucket is found, search returns null, indicating a matching item was not found.

PARTICIPATION ACTIVITY | 13.3.2: Hash table with linear probing: Insert.

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

1) HashInsert(numsTable, item 13)

numsTable:	
0	
1	71
2	22
3	
4	

bucket =

Check Show answer

2) HashInsert(numsTable, item 41)

numsTable:	
0	
1	21
2	
3	
4	

bucket =

Check Show answer

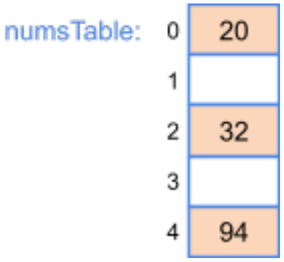
3) HashInsert(numsTable, item 90)

numsTable:	
0	50
1	31
2	
3	
4	4

bucket =

Check Show answer

4) HashInsert(numsTable, item 74)


numsTable:

0	20
1	
2	32
3	
4	94

`bucket =` ______

Check   Show answer

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 | 13.3.3: 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

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 | 13.3.4: Insert with linear probing.

Animation content:

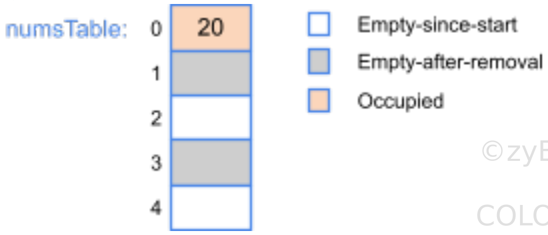
```
undefined
```

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 | 13.3.5: 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)

If unable to drag and drop, refresh the page.

Item 300 **Empty-since-start** **Item 43** **Item 71** **Item 20**

numsTable[0]

numsTable[1]

numsTable[2]

numsTable[3]

numsTable[4]

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 13.3.6: Remove with linear probing.

Animation content:

`undefined`

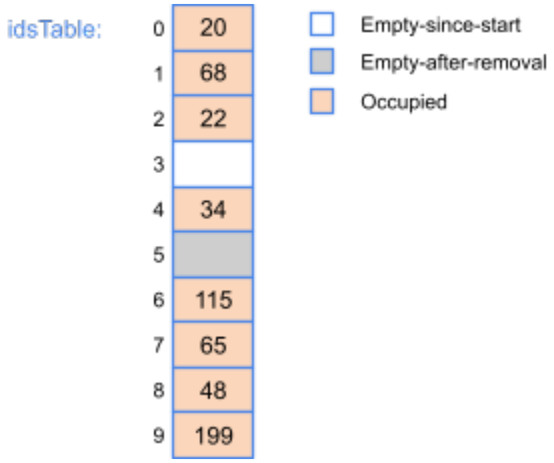
Animation captions:

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.

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 13.3.7: Hash table with linear probing: Remove.

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



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

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

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.

PARTICIPATION ACTIVITY 13.3.8: Search with linear probing.

Animation content:

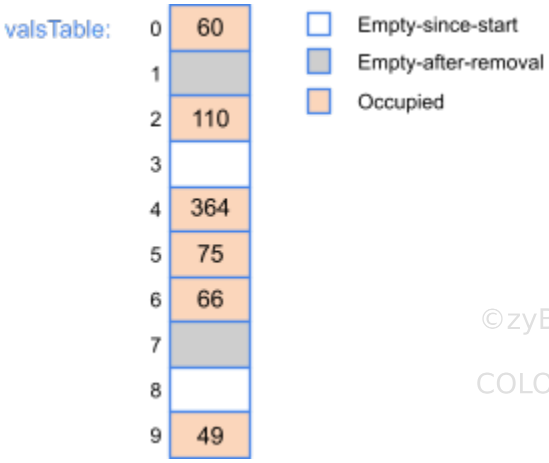
`undefined`

Animation captions:

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.

PARTICIPATION ACTIVITY 13.3.9: Hash table with linear probing: Search.

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



0	60
1	▨
2	110
3	□
4	364
5	75
6	66
7	▨
8	□
9	49

1) HashSearch(valsTable, 75)
probes _____ buckets.

Check Show answer

2) HashSearch(valsTable, 110) probes _____ buckets.

Check Show answer

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

Check Show answer

4) HashSearch(valsTable, 207) probes _____ buckets.

Check Show answer

CHALLENGE ACTIVITY | 13.3.1: Linear probing.

422352.2723990.qx3zqy7

Start

valsTable:

0	
1	41
2	
3	43
4	74
5	
6	
7	
8	
9	

□ Empty-since-start

▨ Empty-after-removal

■ Occupied

Hash table valsTable uses linear probing and a hash function of

HashInsert(valsTable, item 31) inserts item 31 into bucket Ex: 1

HashInsert(valsTable, item 13) inserts item 13 into bucket

HashInsert(valsTable, item 17) inserts item 17 into bucket

1	2	3	4

Check Next

13.4 Quadratic probing

Overview and insertion

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 $(H + c1 * i + c2 * i^2) \bmod (tablesize)$ 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 13.4.1: 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.

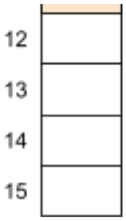
Figure 13.4.1: HashInsert with quadratic probing.

```
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
}
```

PARTICIPATION ACTIVITY 13.4.2: 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.

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



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

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

○ 5

○ 9

○ 11

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 | 13.4.3: Search and removal with quadratic probing: c1 = 1 and c2 = 1.

Animation captions:

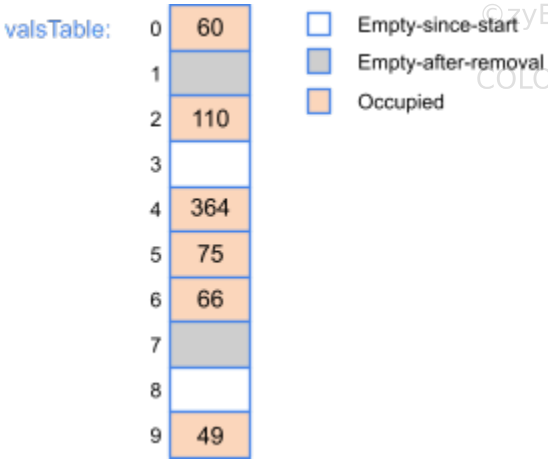
1. 16, 32, and 64 all have a mapped bucket of 0. 32 was inserted first, then 16, then 64.
2. A search for 64 iterates through indices in the probe sequence: 0, 2, then 6.
3. Removal of 32 marks bucket 0 as empty-after removal.
4. A search for 64 after removing item 32 checks the empty-after-removal bucket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

Figure 13.4.2: HashRemove and HashSearch with quadratic probing.

```
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
}
```

PARTICIPATION ACTIVITY | 13.4.4: Hash table with quadratic probing: search and remove.

Consider the following hash table, a hash function of key % 10, and quadratic probing with c1 = 1 and c2 = 1.



1) HashSearch(valsTable, 75) probes _____ buckets.

Check Show answer

2) HashSearch(valsTable, 110) probes _____ buckets.

Check Show answer

3) After removing 66 via HashRemove(valsTable, 66), HashSearch(valsTable, 66) probes _____ buckets.

Check Show answer

PARTICIPATION ACTIVITY | 13.4.5: 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 | 13.4.1: Quadratic hashing.

422352.2723990.qx3zqy7

Start

valsTable:

0	
1	61
2	
3	
4	
5	
6	
7	
8	

□ Empty-since-start

▒ Empty-after-removal

🟧 Occupied

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

HashInsert(valsTable, item 18) inserts item 18 into bucket Ex: 1

HashInsert(valsTable, item 21) inserts item 21 into bucket

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

1	2	3	4

Check Next

13.5 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 h1 and h2, a key's index in the table is computed with the formula $(h1(key) + i * h2(key)) \bmod (tablesize)$. 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 13.5.1: 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 13.5.2: Double hashing.

Given:
hash1(key) = key % 11
hash2(key) = 5 - 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)
HashInsert(valsTable, item 55)
HashInsert(valsTable, item 41)
HashInsert(valsTable, item 63)

1) Item 16

Check Show answer

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 | 13.5.3: Hash table insertion, search, and removal using double hashing.

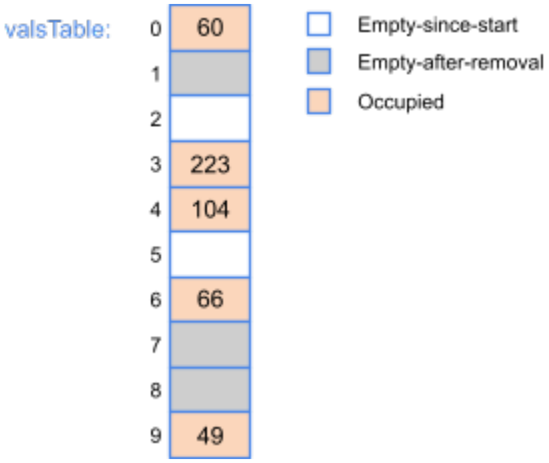
Animation captions:

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 | 13.5.4: 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

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

PARTICIPATION ACTIVITY | 13.5.5: 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.

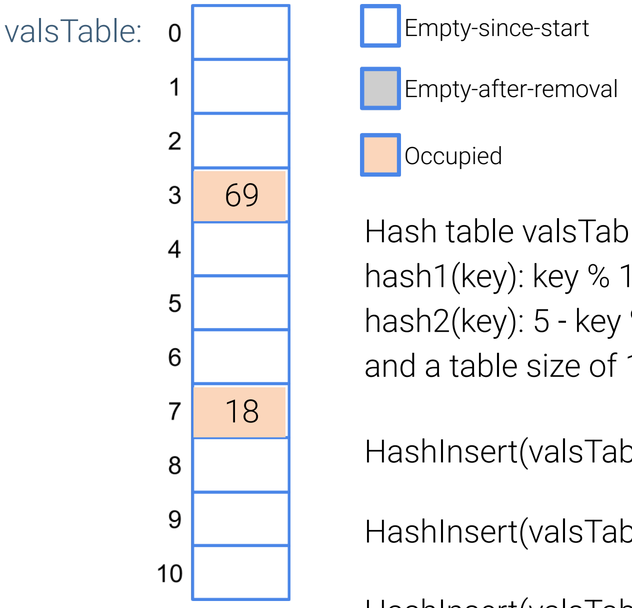
○ True

○ False

CHALLENGE ACTIVITY | 13.5.1: Double hashing.

422352.2723990.qx3zqy7

Start



valsTable:

0	
1	
2	
3	69
4	
5	
6	
7	18
8	
9	
10	

☐ Empty-since-start

▨ Empty-after-removal

🟧 Occupied

Hash table valsTable uses double probing with the hash functior
hash1(key): key % 11
hash2(key): 5 - key % 5
and a table size of 11.

HashInsert(valsTable, item 93) inserts item 93 into bucket Ex: 1

HashInsert(valsTable, item 36) inserts item 36 into bucket

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

1 2 3 4

Check Next

13.6 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 ≥ 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 13.6.1: Hash table resize operation.

Animation content:

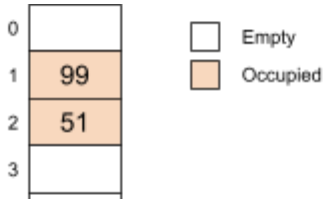
undefined

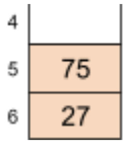
Animation captions:

1. When resizing a hash table with 11 buckets and 7 items, the new size is computed as the next prime number >= 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 13.6.2: Resizing a hash table.

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





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

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 $18/31 = 0.58$. 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 13.6.3: Resizing when a chaining bucket is too large.

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 >= 4.

PARTICIPATION ACTIVITY 13.6.4: Resizing when the load factor is ≥ 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 ≥ 0.6.

PARTICIPATION ACTIVITY 13.6.5: 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 $\lfloor 11/3 \rfloor = 3$ collisions, then the insertion would cause a resize.

PARTICIPATION ACTIVITY 13.6.6: 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

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

○ True

○ False

PARTICIPATION ACTIVITY | 13.6.7: 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 ≤ 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

13.7 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.

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.

Figure 13.7.1: Modulo hash function.

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

PARTICIPATION ACTIVITY 13.7.1: 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

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

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 $\lceil \log_{10} N \rceil$ 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 | 13.7.2: 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

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

Check Show answer

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 $\lceil \log_2 N \rceil$. Ex: For a hash table size of 200, R = 8, then 8 bits are needed for indices 0 to 199.

Figure 13.7.2: 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.

PARTICIPATION ACTIVITY | 13.7.3: 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

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 13.7.3: Multiplicative string hash function.

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

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

   return stringHash % N
}
```

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 13.7.4: 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

1) Initial value = 0
Hash multiplier = 1
String = BAT

Check Show answer

2) Initial value = 0
Hash multiplier = 1
String = TAB

Check Show answer

3) Initial value = 17
Hash multiplier = 3
String = WE

Check Show answer

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.

13.8 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 13.8.1: Direct hash function.

Animation content:

```
undefined
```

Animation captions:

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.

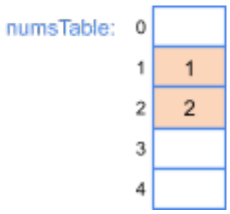
Figure 13.8.1: Direct hashing: Insert, remove, and search operations use item's key as bucket index.

```
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
   }
}
```

PARTICIPATION ACTIVITY 13.8.2: 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)



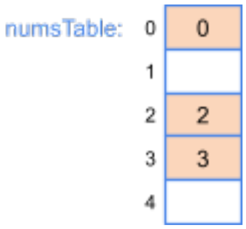
HashInsert(numsTable, item 0)

numsTable:

Check

Show answer

2)



HashRemove(numsTable, 0)

HashInsert(numsTable, item 4)

`numsTable:`

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 13.8.3: 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

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

HashInsert(hashIndex, item -45)

Check Show answer

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

13.9 Hashing Algorithms: Cryptography, Password Hashing

Cryptography

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 13.9.1: Basic encryption: Caeser 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.

PARTICIPATION ACTIVITY 13.9.2: Caesar cipher.

1) What is the result of applying the Caeser cipher with a left shift of 1 to the string "computer"?

○ eqorwvgt

○ dpnqvufs

○ bnlotsdq

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 Caeser cipher were implemented such that strings were restricted to only lower-case alphabet characters, how many distinct ways could a message be encrypted?

○ 26

○ 52

○ Length of the message

PARTICIPATION ACTIVITY 13.9.3: Cryptography.

1) Encryption and decryption are synonymous.

○ True

○ False

2) Cryptography is used heavily in internet communications.

○ True

○ False

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

○ True

○ False

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 13.9.4: 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.

PARTICIPATION ACTIVITY 13.9.5: 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

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 13.9.6: Password hashing function.

Animation captions:

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.

PARTICIPATION ACTIVITY 13.9.7: Password hashing function.

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.

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

13.10 Map: HashMap

Map container

A programmer may wish to lookup values or elements based on another value, such as looking up an employee's record based on an employee ID. The ***Map*** interface within the Java Collections Framework defines a Collection that associates (or maps) keys to values. The statement `import java.util.HashMap;` enables use of a HashMap.

The HashMap type is an ADT implemented as a generic class (discussed elsewhere) that supports different types of keys and values. Generically, a HashMap can be declared and created as `HashMap<K, V> hashMap = new HashMap<K, V>();` where K represents the HashMap's key type and V represents the HashMap's value type.

The put() method associates a key with the specified value. If the key does not already exist, a new entry within the map is created. If the key already exists, the old value for the key is replaced with the new value. Thus, a map associates at most one value for a key.

The get() method returns the value associated with a key, such as `statePopulation.get("CA")`.

PARTICIPATION ACTIVITY | 13.10.1: A HashMap allows a programmer to map keys to values.

Animation captions:

1. The put() method associates a key with the specified value.
2. The get() method returns the value associated with a key.

PARTICIPATION ACTIVITY | 13.10.2: HashMap's put() method replaces the value associated with a key.

Animation captions:

1. Using put() with an existing key replaces the value associated with that key.

PARTICIPATION ACTIVITY | 13.10.3: Basic HashMap operations: put() and get().

Given the following code that creates and initializes a HashMap:

```
HashMap<String, Integer> playerScores = new HashMap<String, Integer>();
playerScores.put("Jake", 14);
playerScores.put("Pat", 26);
playerScores.put("Hachin", 60);
playerScores.put("Michiko", 21);
playerScores.put("Pat", 31);
```

1) Write a statement to add a mapping for a

player named Kira with a score of 1.

Check Show answer

2) Write a statement to assign Hachin's score to a variable named highScore. □

Check Show answer

3) What value will playerScores.get("Pat") return. □

Check Show answer

4) Write a single statement to update Jake's score to the value 34. □

Check Show answer

Determining if a key exists

If the map does not contain the specified key, the get() method returns ***null***. A programmer can use the containsKey() method to check if a map contains the specific key. In the program above, `statePopulation.containsKey("NY")` would return false.

zyDE 13.10.1: Use containsKey() to check if the HashMap contains the use specified key.

This program uses a HashMap to associate a race distance in kilometers with a runn time. If a race distance does not exist, the program prints "null". Modify the program t containsKey() method to check if the runner has run a race of the specified distance, display a message of "No race of the specified distance exists."

RunDistTimeMap.java

Load default ter

```
import java.util.HashMap;
import java.util.Scanner;

public class RunDistTimeMap {
   public static void main (String[] args) {
      HashMap<Integer, Double> raceTimes = new HashMap<Integer, Double>();
      Scanner scnr = new Scanner(System.in);
      int userDistKm;

      raceTimes.put(5, 23.14);
      raceTimes.put(15, 78.5);
      raceTimes.put(25, 120.75);

      System.out.println("Enter race distance in km (0 to exit): ");
      userDistKm = scnr.nextInt();

      while(userDistKm != 0) {
```

```
5 15 10 0
```

Run

PARTICIPATION ACTIVITY | 13.10.4: Determining if map contains a key.

Given the code below, determine the result of each expression.

```
HashMap<Integer, Double> raceTimes = new HashMap<Integer, Double>();

raceTimes.put(5, 23.14);
raceTimes.put(15, 78.5);
raceTimes.put(25, 120.75);
```

1) raceTimes.containsKey(10)

○ true

○ false

2) raceTimes.containsKey(5)

○ true

○ false

3) raceTimes.get(7)

○ 0

○ null

Common HashMap methods

The HashMap class implements several methods, defined in the Map interface, for accessing and modifying entries within a map.

Table 13.10.1: Common HashMap methods.

put()	`put(key, value)`
	Associates key with specified value. If key already exists, replaces previous value with specified value.
		// Map originally empty
		`exMap.put("Tom", 14);`
		// Map now: Tom->14,
		`exMap.put("John", 86);`
		// Map now: Tom->14, John->86

putIfAbsent()	`putIfAbsent(key, value)`
	Associates key with specified value if the key does not already exist or is mapped to null.
		// Assume Map is: Tom->14, John->86
		`exMap.putIfAbsent("Tom", 20);`
		// Key "Tom" already exists. Map is unchanged.
		`exMap.putIfAbsent("Mary", 13);`
		// Map is now: Tom->14, John->86, Mary->13

get()	`get(key)` Returns the value associated with key. If key does not exist, return null.	*// Assume Map is: Tom->14, John->86, Mary->13* `exMap.get("Tom")` *// returns 14* `exMap.get("Bob")` *// returns null*

containsKey()	`containsKey(key)`
	Returns true if key exists, otherwise returns false.
		// Assume Map is: Tom->14, John->86, Mary->13
		`exMap.containsKey("Tom")`
		// returns true
		`exMap.containsKey("Bob")`
		// returns false

containsValue()	`containsValue(value)`
	Returns true if at least one key is associated with the specified value, otherwise returns false.
		// Assume Map is: Tom->14, John->86, Mary->13
		`exMap.containsValue(86)`
		// returns true (key "John" associated with value 86)
		`exMap.containsValue(17)`
		// returns false (no key associated with value 17)

	`remove(key)`	*// Assume Map is: Tom->14, John->86,*

remove()	`remove(key)` Removes the map entry for the specified key if the key exists.	*Mary->13* `exMap.remove("John");` *// Map is now: Tom->14, Mary->13*
clear()	`clear()` Removes all map entries.	*// Assume Map is: Tom->14, John->86, Mary->13* `exMap.clear();` *// Map is now empty*
keySet()	`keySet()` Returns a Set containing all keys within the map.	*// Assume Map is: Tom->14, John->86, Mary->13* `keys = exMap.keySet();` *// keys contains: "Tom", "John", "Mary"*
values()	`values()` Returns a Collection containing all values within the map.	*// Assume Map is: Tom->14, John->86, Mary->13* `values = exMap.values();` *// values contains: 14, 86, 13*

PARTICIPATION ACTIVITY 13.10.5: Common HashMap methods.

Given the following code that creates and initializes a HashMap:

```
HashMap<String, Integer> exMap = new HashMap<String, Integer>();
exMap.put("Tom", 14);
exMap.put("John", 26);
exMap.put("Mary", 13);
exMap.put("Hans", 90);
exMap.put("Franz", 88);
```

Which of the following operations modify the HashMap?

1) exMap.putIfAbsent("Mary", 17);

○ True

○ False

2) exMap.putIfAbsent("Frederique", 36);

○ True

○ False

3) exMap.remove("Tom");

○ True

○ False

4) exMap.remove("john");

○ True

○ False

5) exMap.clear();

○ True

○ False

CHALLENGE ACTIVITY 13.10.1: Map: HashMap.

422352.2723990.qx3zqy7

Start

Type the program's output

```
import java.util.HashMap;

public class AirportCodes {
   public static void main (String[] args) {
      HashMap<String, String> airportCode = new HashMap<String, String>();

      airportCode.put("GRX", "Granada, Spain");
      airportCode.put("IWJ", "Iwami, Japan");
      airportCode.put("LNZ", "Linz, Austria");

      System.out.print("IWJ: ");
      System.out.println(airportCode.get("IWJ"));

      airportCode.put("LNZ", "Lisbon, Portugal");

      System.out.print("LNZ: ");
      System.out.println(airportCode.get("LNZ"));
   }
}
```

1 2 3

Check Next

HashMap vs. TreeMap

HashMap and TreeMap are ADTs implementing the Map interface. Although both HashMap and TreeMap implement a Map, a programmer should select the implementation that is appropriate for the intended task. A HashMap typically provides faster access but does not guarantee any ordering of the keys, whereas a TreeMap maintains the ordering of keys but with slightly slower access. This material uses the HashMap class, but the examples above can be modified to use TreeMap.

Exploring further:

- HashMap from Oracle's Java documentation.
- TreeMap from Oracle's Java documentation.
- Java Collections Framework Overview from Oracle's Java documentation.

13.11 Set: HashSet

This section has been set as optional by your instructor.

Set container

The ***Set*** interface defined within the Java Collections Framework defines a Collection of unique elements. The Set interface supports methods for adding and removing elements, as well as querying if a set contains an element. For example, a programmer may use a set to store employee names and use that set to determine which customers are eligible for employee discounts.

The HashSet type is an ADT implemented as a generic class that supports different types of elements. A HashSet can be declared and created as `HashSet<T> hashSet = new HashSet<T>();` where T represents the HashSet's type, such as Integer or String. The statement `import java.util.HashSet;` enables use of a HashSet within a program.

PARTICIPATION ACTIVITY 13.11.1: A HashSet's add(), remove(), and contains() methods add an item, remove an item, and check if an item exists within the set.

Animation captions:

1. The add() method adds an item such as a String to a set.
2. The contains() method returns true if an item is in the set, and otherwise returns false.
3. The remove() method removes an item from a set.

PARTICIPATION ACTIVITY 13.11.2: HashSet operations: add(), remove(), and contains().

Given the following code that creates and initializes a HashSet:

```
HashSet<Integer> employeeIDs = new HashSet<Integer>();
employeeIDs.add(1001);
employeeIDs.add(1002);
employeeIDs.add(1003);
```

1) Write a statement that adds an employee ID 1337.

Check   Show answer

2) Write a statement that removes the employee ID 1002.

Check   Show answer

3) What value will employeeIDs.contains(1001) return?

Check   Show answer

add() and remove() methods

The add() method does not add duplicate elements to a set. If a programmer tries to add a duplicate element, the add() method fails and returns false. Otherwise, add() returns true.

The remove() method only removes elements that exist within the set. If the element exists, the remove() method removes the element and returns true. Otherwise, remove() returns false.

PARTICIPATION ACTIVITY | 13.11.3: A HashSet's add() and remove() methods return a boolean to indicate the operation's failure or success.

Animation captions:

1. A set does not contain duplicate items. The add() method adds an item only if the item does not already exist. add() returns true if item added, else returns false.
2. The remove() method returns true if item removal was successful, otherwise returns false.

PARTICIPATION ACTIVITY | 13.11.4: Return value of HashSet's add() and remove() methods.

Given the following code that creates and initializes a HashSet:

```
HashSet<Character> guessedLetters = new HashSet<Character>();
guessedLetters.add('e');
guessedLetters.add('t');
guessedLetters.add('a');
```

1) What value will guessedLetters.remove('s') return?

Check Show answer

2) What value will guessedLetters.add('e') return?

Check Show answer

3) Write a single statement that adds to guessedLetters the variable favoriteLetter only if favoriteLetter does not already exist in the set.

Check Show answer

zyDE 13.11.1: Use only remove() to check if the user's guess is in the set.

The program below uses a HashSet to store the numbers a user has to guess to win. program uses both contains() and remove() to remove correct guesses from the set, game ends when user guesses all numbers (i.e., the set is empty). Modify the prograı use remove(), not contains(). Hint: consider the return value of remove().

GuessTheNumbers.java

Load default ter

```
import java.util.HashSet;
import java.util.Scanner;

public class GuessTheNumbers {
   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      HashSet<Integer> numbersToGuess = new HashSet<Integer>();
      int userGuess;

      numbersToGuess.add(3);
      numbersToGuess.add(5);
      numbersToGuess.add(1);

      System.out.print("Enter a number between 1 to 10 (0 to exit): ");
      userGuess = scnr.nextInt();

      while (userGuess != 0) {
```

```
1 2 5 3
```

Run

Common HashSet operations

The HashSet class implements several methods, defined in the Set interface, for modifying and querying the status of the set.

Table 13.11.1: Common HashSet methods.

add()	`add(element)`
	If element does not exist, adds element to the set and returns true. If element already exists, returns false.
		`// Set originally empty`
		`exSet.add("Kasparov"); // returns true (element "Kasparov" does not exist)`
		`// Set is now: Kasparov`
		`exSet.add("Kasparov"); // returns false (element "Kasparov" already exists)`
		`// Set is unchanged`

remove()	`remove(element)`
	If element exists, removes element from the set and returns true. If the element does not exist, returns false.
		`// Assume Set is: Kasparov, Fisher`
		`exSet.remove("Fisher");  // returns true (element "Fisher" exists)`
		`// Set is now: Kasparov`
		`exSet.remove("Carlsen"); // returns false (element "Carlsen" does not exist)`
		`// Set is unchanged`

contains()	`contains(element)`
	Returns true if element exists, otherwise returns false.
		`// Assume Set is: Kasparov, Fisher, Carlsen`
		`exSet.contains("Carlsen")  // returns true`
		`exSet.contains("Anand")  // returns false`

size()	`size()`
	Returns the number of elements in the set.
		`// Set originally empty`
		`exSet.size(); // returns 0`
		`exSet.add("Nakamura");`
		`exSet.add("Carlsen");`
		`// Set is now: Nakamura, Carlsen`
		`exSet.size(); // returns 2`

CHALLENGE ACTIVITY 13.11.1: Using a HashSet to define unique elements.

422352.2723990.qx3zqy7

Start

Type the program's output

```
import java.util.Scanner;
import java.util.HashSet;

public class BooksSet {
   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      HashSet<String> books = new HashSet<String>();
      String userInput;

      userInput = scnr.nextLine();
      while (userInput.compareTo("done") != 0) {
         if (books.add(userInput)) {
            System.out.println("a");
         }
         else {
            System.out.println("n");
         }
         userInput = scnr.nextLine();
      }

      System.out.println("Size: " + books.size());
   }
}
```

Input

```
The Great Gatsby
The Great Gatsby
The Great Gatsby
Ulysses
The Stranger
War and Peace
The Iliad
done
```

Output

1 2

Check Next

HashSet vs. TreeSet

HashSet and TreeSet are ADTs implementing the Set interface. Although both HashSet and TreeSet implement a Set, a programmer should select the implementation that is appropriate for the intended task. A HashSet typically provides faster access but does not guarantee any ordering of the elements, whereas a TreeSet maintains the ordering of elements but with slightly slower access. In this material, we use the HashSet class, but the examples can be modified to TreeSet.

Exploring further:

- HashSet from Oracle's Java documentation.
- TreeSet from Oracle's Java documentation.
- Java Collections Framework Overview from Oracle's Java documentation.

13.12 LAB: Student grades (HashMap)

Given a HashMap pre-filled with student names as keys and grades as values, complete main() by reading in the name of a student, outputting their original grade, and then reading in and outputting their new grade.

Ex: If the input is:

```
Quincy Wraight
73.1
```

the output is:

```
Quincy Wraight's original grade: 65.4
Quincy Wraight's new grade: 73.1
```

422352.2723990.qx3zqy7

LAB ACTIVITY | 13.12.1: LAB: Student grades (HashMap) | 0 / 10

StudentGrades.java

Load default template...

```java
import java.util.Scanner;
import java.util.HashMap;

public class StudentGrades {

   public static void main (String[] args) {
      Scanner scnr = new Scanner(System.in);
      String studentName;
      double studentGrade;

      HashMap<String, Double> studentGrades = new HashMap<String, Double>();

      // Students's grades (pre-entered)
      studentGrades.put("Harry Rawlins", 84.3);
      studentGrades.put("Stephanie Kong", 91.0);
      studentGrades.put("Shailen Tennyson", 78.6);
      studentGrades.put("Quincy Wraight", 65.4);
```

Develop mode **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → StudentGrades.java (Your program) →

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin
working on this zyLab.
```

13.13 Lab 23 - Build a Hash: Hashing Algorithms

Lab 23: Build a Hash - Hashing Algorithms

Hashing Algorithms

This lab has a single Java file you can download below if you would like to code in an IDE.

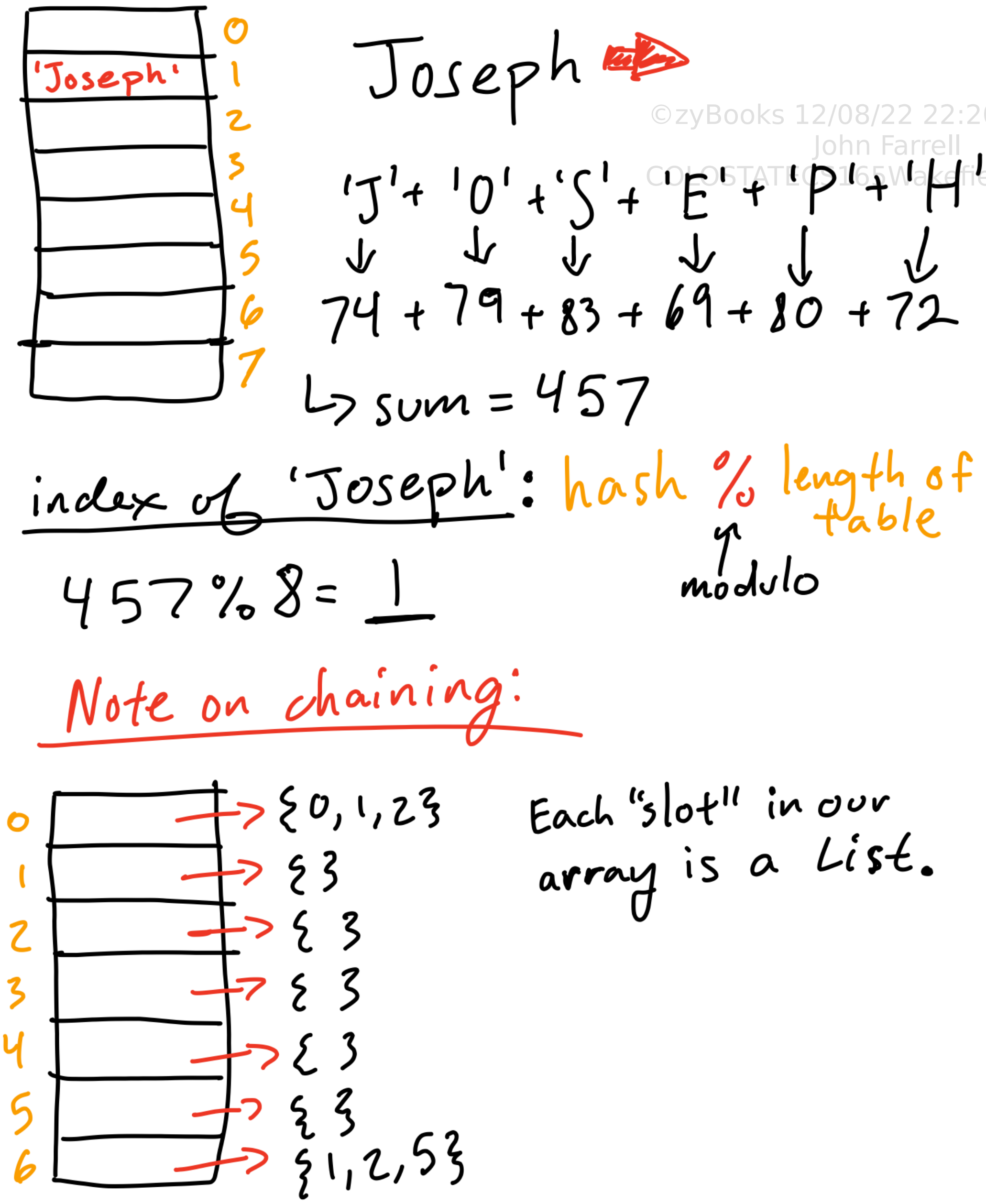
Hashing is a very important core concept of computer science because it allows us to turn arbitrary data like strings and even objects into numbers which can allow us to use data structures like hash tables.

Hash Tables - Overview

There are a myriad of ways to hash and the ones presented in this lab are quite simple and easy to implement. In the real world hashes are often much more complicated and depending on whether they are to be used for cryptography or a data structure. Below are some links to some good resources about more complicated hashes if you are curious:

- HMC CS hash function
- Wikipedia Hash Function
- Wikipedia Secure Hash Algorithms

This diagram shows on the top how a simple hash of a string may work. The bottom of the diagram explains how the hash table in this lab is implemented with chaining. The next lab will explore other collision strategies in much more depth.



Joseph

'J'+'O'+'S'+'E'+'P'+'H'

74 + 79 + 83 + 69 + 80 + 72

↳ sum = 457

index of 'Joseph': hash % length of table

457 % 8 = 1

Note on chaining:

Each "slot" in our array is a List.

Completing the Code

HINT: Remember that the ASCII value of a character can be retrieved by simply casting that character to an int.

First, Complete the three hash functions primeHash, myHash, and asciiHash in any order.

Then, complete the insert method.

Reminder: The hash could become negative if the string is very long and the hash overflows the max value of an int so make sure to return the absolute value of the hash.

IMPORTANT NOTE:

numBuckets is not initialized in the constructor, so if you choose to use it then please initialize it to an appropriate value.

Testing

Run the main method and you should get this output if your code is correct:

```
Testing asciiHash...
Expected: 387, Actual: 387

Testing primeHash...
Expected: 15887, Actual: 15887

Testing myHash...
Expected: 1296, Actual: 1296

TABLE 0
bucket 0: [Noah, Oliver, Harper, Mason]
bucket 1: [James]
bucket 2: [Liam, Evelyn]
bucket 3: []
bucket 4: [William, Benjamin, Charlotte, Mia, Lucas]

TABLE 1
bucket 0: []
bucket 1: [James]
bucket 2: [William, Benjamin, Charlotte, Mia, Lucas]
bucket 3: [Liam, Evelyn]
bucket 4: [Noah, Oliver, Harper, Mason]

TABLE 2
bucket 0: []
bucket 1: [William, Charlotte, Mia, Harper, Mason]
bucket 2: [James, Oliver, Evelyn]
bucket 3: [Liam, Benjamin, Lucas]
bucket 4: [Noah]
```

Note: the tests will be similar to the tests in main.

Submission

Submit your code here in zyLabs in Submit mode

422352.2723990.qx3zqy7

LAB ACTIVITY 13.13.1: Lab 23 - Build a Hash: Hashing Algorithms 0 / 7

Downloadable files

HashTable.java **Download**

HashTable.java

```
Loading latest submission...
```

Develop mode **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → HashTable.java (Your program) →

Program output displayed here

Coding trail of your work What is this?

Retrieving signature

13.14 Lab 24 - Hash: Handle Collisions

Lab 24: Hash - Handle Collisions

In the last lab, you wrote different types of hash functions. In this lab, you will learn how to handle different types of collisions. Remember that a collision is when the hash function maps two different keys (or more) to the same location in the hash table. You can just add multiple items to the same location like we did in the last lab. This is known as chaining. We are implementing that again in this lab so that you can compare it to the other ways to handle collisions. Chaining can make searching for a specific item take longer as the number of items in that bucket increases. This is why you will be implementing different ways to handle collisions as well.

Hashing

Here is the jar: L24.jar

The following files are included in this lab:

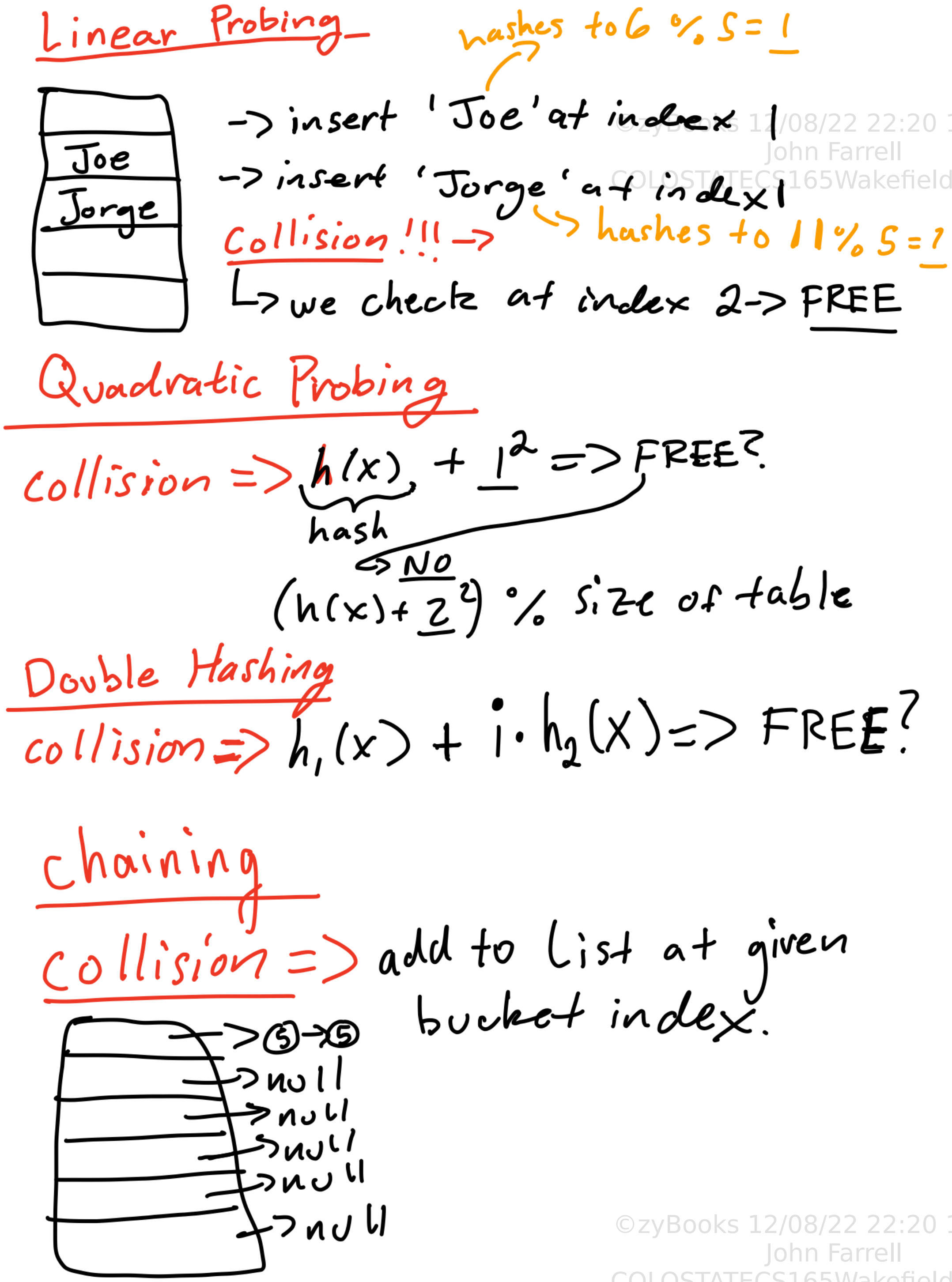
L24

```
├── ChainedTable.java
├── DoubleTable.java
├── ITable.java
├── LinearTable.java
├── QuadraticTable.java
├── LinearTable.java
├── TestTables.java *
```

***** This is the main in zyLabs. To run your program, you must enter in the type of table you are running: linear, quadratic, double, or chained.

Ways to Manage Collisions

There are four main ways to deal with collisions in hash tables that we will look at here: Linear Probing, Quadratic Probing, Double Hashing, and Chaining.



Linear Probing

hashes to 6 % 5 = 1

-> insert 'Joe' at index 1

-> insert 'Jorge' at index 1 <- hashes to 11 % 5 = 1

Collision!!! ->

↳ we check at index 2 -> FREE

Quadratic Probing

collision => $\underbrace{h(x)}_{\text{hash}} + 1^2$ => FREE?

NO

$(h(x) + 2^2)$ % size of table

Double Hashing

collision => $h_1(x) + i \cdot h_2(x)$ => FREE?

chaining

collision => add to list at given bucket index.

Completing the Code

You may finish the classes in any order you wish. Each Class implements a different chaining strategy. There are ample descriptions in the documentation above each function describe how one would implement them. You must complete these two functions in all classes:

1. insert()

2. search()

Testing the Different Tables

The main class TestTables.java will test each table based on the input you give it. To run this, you will need to enter in the name of the type of table you would like to test: chained, double, linear, or quadratic.

Testing LinearTable

Input 'linear' when you run your program and you should get this output if your code is correct:

```
Bucket 0: Harper
Bucket 1: Lucas
Bucket 2: Liam
Bucket 3: William
Bucket 4: Benjamin
Bucket 5: Noah
Bucket 6: Oliver
Bucket 7: Charlotte
Bucket 8: Mia
Bucket 9: James
Bucket 10: Evelyn

Testing search(Liam)...
PASSED -> Expected: true, Actual: true
Testing search(Joshua)...
PASSED -> Expected: false, Actual: false
```

Testing QuadraticTable

Input 'quadratic' when you run your program and you should get this output if your code is correct:

```
Bucket 0: null
Bucket 1: Evelyn
Bucket 2: Liam
Bucket 3: William
Bucket 4: Benjamin
Bucket 5: Noah
Bucket 6: Oliver
Bucket 7: Charlotte
Bucket 8: Mia
Bucket 9: James
Bucket 10: Harper

Testing search(Liam)...
PASSED -> Expected: true, Actual: true
Testing search(Joshua)...
PASSED -> Expected: false, Actual: false
```

Testing DoubleTable

Input 'double' when your run your program and you should get this output if your code is correct:

```
Bucket 0: Charlotte
Bucket 1: Lucas
Bucket 2: Liam
Bucket 3: James
Bucket 4: Benjamin
Bucket 5: Noah
Bucket 6: Oliver
Bucket 7: Harper
Bucket 8: Mia
Bucket 9: William
Bucket 10: Evelyn

Testing search(Liam)...
PASSED -> Expected: true, Actual: true
Testing search(Joshua)...
PASSED -> Expected: false, Actual: false
```

Testing ChainedTable

Input 'chained' when you run your program and you should get this output if your code is correct:

```
Bucket 0: []
Bucket 1: [Mason]
Bucket 2: [Liam, William]
Bucket 3: [Lucas]
Bucket 4: [Benjamin]
Bucket 5: [Noah]
Bucket 6: [Oliver, Charlotte]
Bucket 7: []
Bucket 8: [Mia, Evelyn]
Bucket 9: [James]
Bucket 10: [Harper]

Testing search(Liam)...
PASSED -> Expected: true, Actual: true
Testing search(Joshua)...
PASSED -> Expected: false, Actual: false
```

422352.2723990.qx3zqy7

LAB ACTIVITY | 13.14.1: Lab 24 - Hash: Handle Collisions | 0 / 7

Downloadable files

`ChainedTable.java` , `DoubleTable.java` , `ITable.java` , `LinearTable.java` , `QuadraticTable.java` , and `TestTables.java`

Download

Current file: **ChainedTable.java** ▾ Load default template...

```java
import java.util.ArrayList;
import java.util.List;

public class ChainedTable implements ITable{
    private ArrayList<List<String>> table;
    private int numBuckets;
    private int numElements;

    public ChainedTable(int numBuckets){
        this.numElements = 0;
        table = new ArrayList<List<String>>();
        for(int i = 0; i < numBuckets; i++){
            table.add(new ArrayList<String>());
        }
    }
    /**
     * TODO - Complete This Function
```

Develop mode **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → **ChainedTable.java** (Your program) —

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin
working on this zyLab.
```