5.1 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 5.1.1: 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 5.1.2: 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.

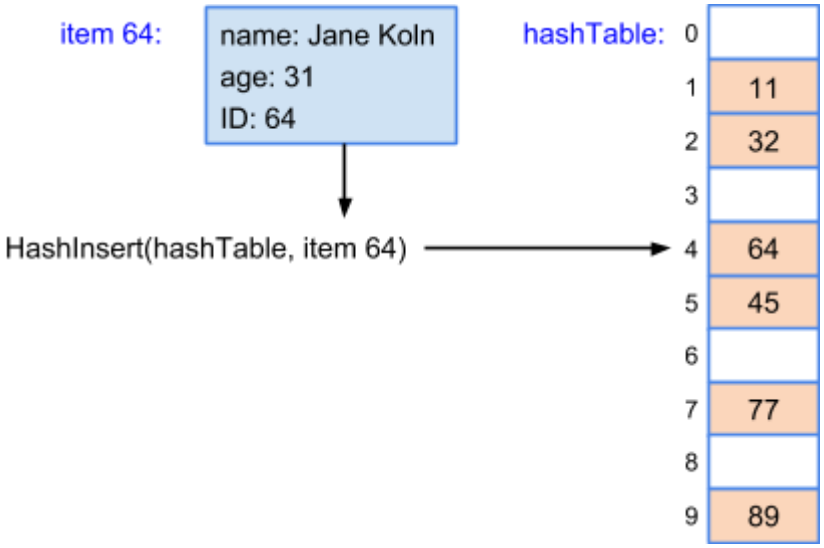
- O(1)
- O(N)
- O(log N)

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 5.1.3: 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?

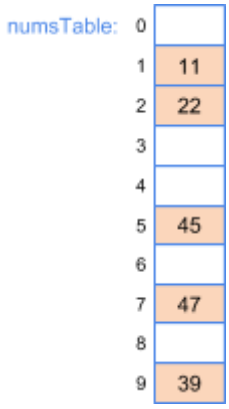
Check Show answer

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 5.1.4: Hash table search efficiency.

Consider a modulo hash function and the following hash table.



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 5.1.5: 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

How was this section? Provide feedback

5.2 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 5.2.1: Hash table with chaining.

Animation content:

`undefined`

Animation captions:

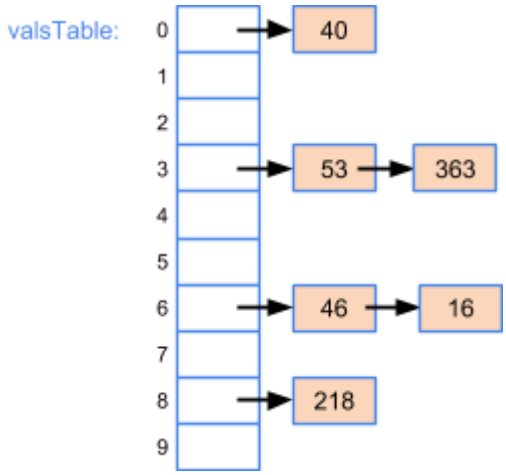
1. A hash table with chaining uses a list for each bucket. The insert operation first uses the item's key to determine the mapped bucket, and then inserts the item in that bucket's list.
2. A bucket may store multiple items with different keys that map to the same bucket. If collisions occur, items are 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 5.2.1: 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 5.2.2: 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: _____

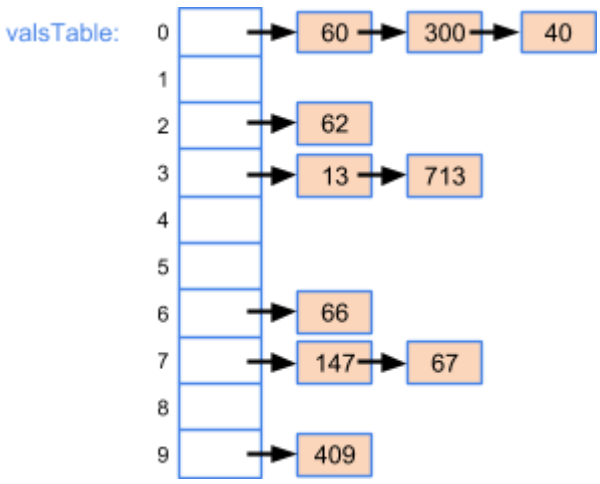
Check Show answer

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

Check Show answer

PARTICIPATION ACTIVITY 5.2.3: 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

How was this section? Provide feedback

5.3 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 5.3.1: Hash table with linear probing.

Animation captions:

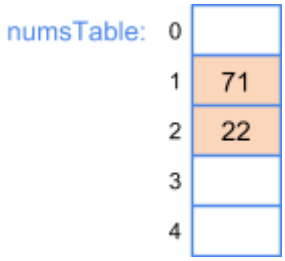
1. During an insert, if a bucket is not empty, a collision occurs. Using linear probing, inserts wil linearly probe buckets until an empty bucket is found.

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 5.3.2: Hash table with linear probing: Insert.

Given hash function of key % 10, determine the insert location for each item.

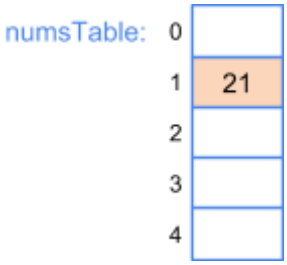
1) HashInsert(numsTable, item 13)



bucket =

Check Show answer

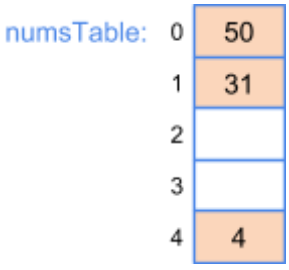
2) HashInsert(numsTable, item 41)



bucket =

Check Show answer

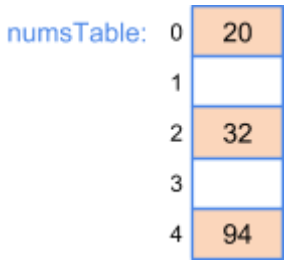
3) HashInsert(numsTable, item 90)



bucket =

Check Show answer

4) HashInsert(numsTable, item 74)



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 5.3.3: 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.

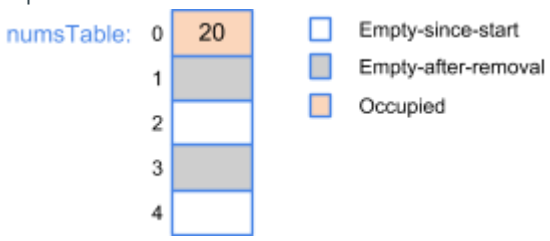
PARTICIPATION ACTIVITY 5.3.4: Insert with linear probing.

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 5.3.5: 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)

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

---

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 5.3.6: Remove with linear probing.

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-star 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 5.3.7: Hash table with linear probing: Remove.

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

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 5.3.8: Search with linear probing.

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 bucke 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 5.3.9: Hash table with linear probing: Search.

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

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

How was this section? Provide feedback

5.4 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 5.4.1: Hash table insertion using quadratic probing: c1 = 1 and c2 = 1.

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 5.4.1: 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 5.4.2: 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
12	
13	
14	
15	

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 5.4.3: 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 5.4.2: 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 5.4.4: 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 5.4.5: 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 an empty-since-start bucket or 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

How was this section? Provide feedback

5.5 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 5.5.1: Hash table insertion using double hashing.

Animation content:

```
undefined
```

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 5.5.2: 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 5.5.3: Hash table insertion, search, and removal using double hashing.

Animation content:

`undefined`

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-afte removal bucket, and item 88 is inserted in that bucket.

PARTICIPATION ACTIVITY 5.5.4: 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.

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 5.5.5: 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

How was this section? Provide feedback

5.6 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 5.6.1: Modulo hash function.

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

PARTICIPATION ACTIVITY 5.6.1: 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 $\lceil \log N \rceil$ 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 5.6.2: 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 5.6.2: 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 5.6.3: 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 5.6.3: Multiplicative string hash function.

```
HashMutliplicative(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 5.6.4: 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.

How was this section? Provide feedback

5.7 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 5.7.1: Direct hash function.

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.

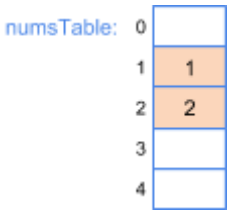
Figure 5.7.1: Direct hashing: Insert, remove, and search operations use item's key as bucket index.

```
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 5.7.2: 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)

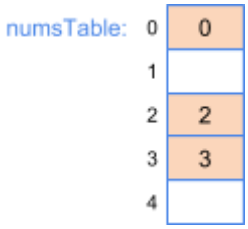


HashInsert(numsTable, item 0)

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 5.7.3: 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

How was this section? Provide feedback

5.8 Hashing Algorithms: Cryptography, Password H

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 5.8.1: Basic encryption: Caeser cipher.

Animation content:

undefined

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 5.8.2: 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 5.8.3: 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 5.8.4: A hash value can help identify corrupted data downloaded from the internet.

Animation content:

```
undefined
```

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 5.8.5: 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 5.8.6: Password hashing function.

Animation content:

`undefined`

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 t 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 5.8.7: 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

How was this section? **Provide feedback**