Hashing

Notes from Charlie Obimbo and revised by Yan Yan.

Contents

1. Hash Tables
2. Chaining
3. Linear Probing
4. Quadratic Probing
5. Double Hashing

Learning Objectives

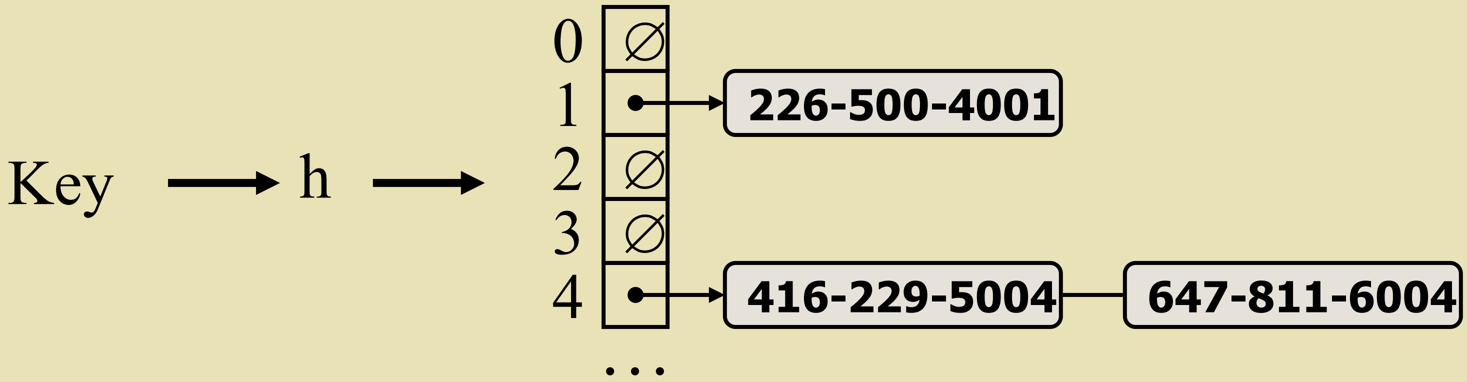
1. Define hash tables
2. List different approaches to solve collisions in hash tables
3. Calculate the indices of given elements using different hash functions
4. Implement common operations in hash tables

Hash Tables

- 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).
- An item's **key** is the value used to map to an index.
- Each hash table array element is called a **bucket**. A **hash function** computes a bucket index from the item's key.

Hash Tables

- Example in the textbook 6.1.1
- Example of storing phone numbers

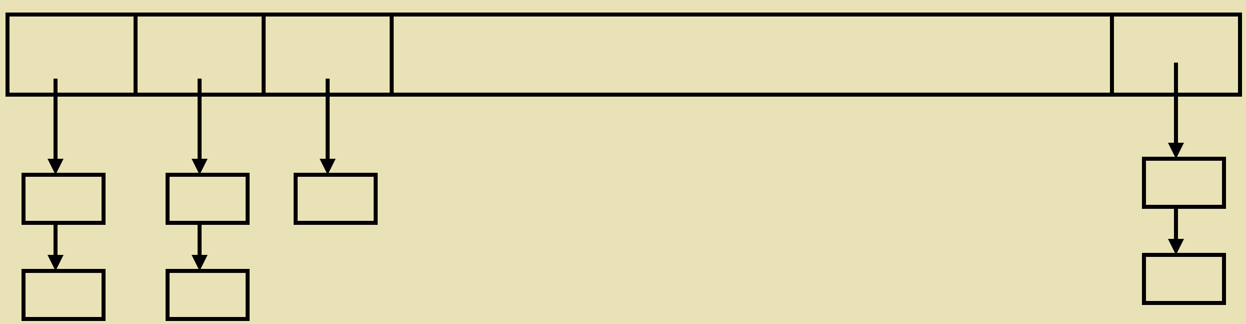


Collisions

- 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.
- **Chaining** is a collision resolution technique where each bucket has a list of items.

Collision resolution using linked Lists:

- Dynamically allocate space.
- Easy to insert/delete an item
- Need a link for each node in the hash table.



Why Hash Tables?

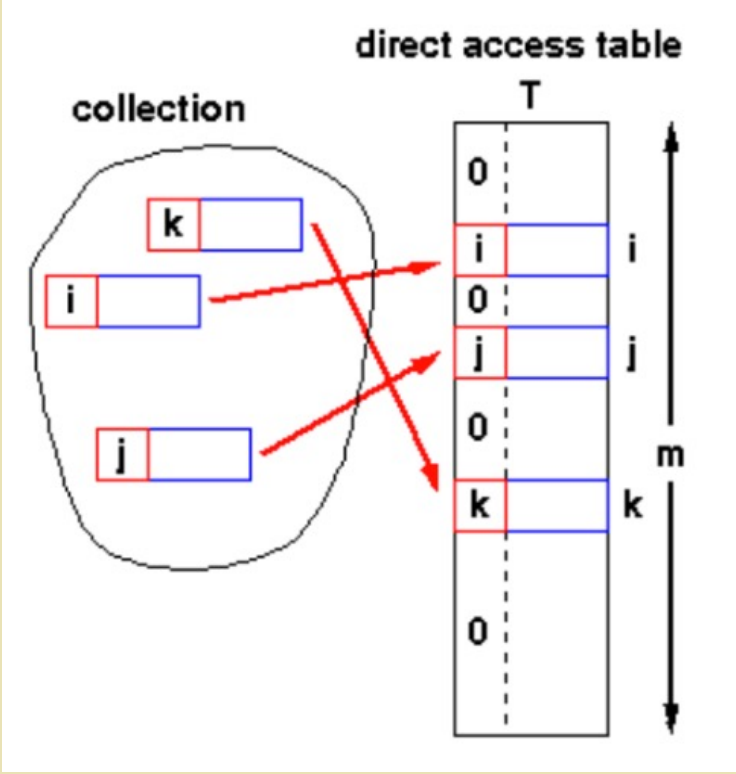
- All search structures so far
  - Relied on a comparison operation
  - Performance $O(n)$ or $O(\log n)$
- Assume we have a function
  - $f(key) \rightarrow integer$

    i.e. function that maps a key to an integer
- What performance might we expect now?

$O(1)$

Hash Tables – Structure

- Simplest case:
  - Assume items have integer keys in the range **1 .. *m***
  - Use the value of the key itself to select a slot (**bucket**) in a **direct access table** to **store** the item
- To **search** for an item with key, *k*, just look in slot *k*
  - If there's an item there, you've found it
  - If the tag is 0, it's missing.
- Constant time, **O(1)**

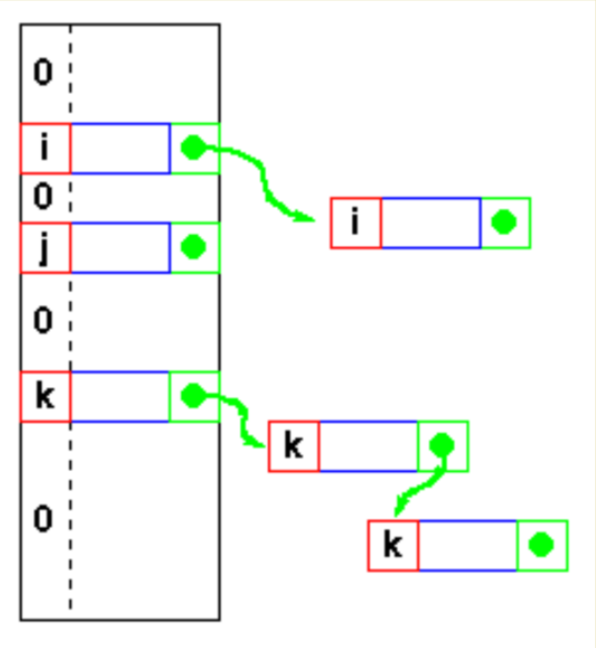


Hash Tables - Constraints

- **Constraints**
  - Keys must be unique
  - Keys must be integers
  - For storage efficiency, keys must be **dense** in the range
  - If they're **sparse** (lots of gaps between values), a lot of (unnecessary) space is used to obtain speed

Hash Tables - Relaxing the constraints

- "Keys are integers"
  - Need a ***hash function***
  - h(key) → integer
    i.e. one that maps a key of a different type (e.g. char) to an integer
  - Applying this function to the key produces an address
  - If h( ) maps each key to a unique integer in the range 0 .. m-1, then search is O(1)



An Example: Perfect Hash

- suppose: **MagicNumber = 15**

```
int h(String s) {
    return ((s[0] + s[1])% MagicNumber);
}
```

- suppose:

```
typedef struct {
String name;
int numMoons;
double sunDistance;
} planet;
planet solarSystem[MagicNumber];
```

An Example: Perfect Hash

– Suppose:

solarSystem[h("Mercury")] = {"Mercury", 0, 36.0};

solarSystem[h("Venus")] = {"Venus", 0, 67.27};

solarSystem[h("Earth")] = {"Earth", 1, 93.0};

solarSystem[h("Mars")] = {"Mars", 2, 141.71};

solarSystem[h("Jupiter")] = {"Jupiter", 16, 483.88};

solarSystem[h("Saturn")] = {"Saturn", 12, 887.14};

solarSystem[h("Uranus")] = {"Uranus", 5, 1783.98};

solarSystem[h("Neptune")] = {"Neptune", 2, 2795};

solarSystem[h("Pluto")] = {"Pluto", 1, 3675};

– Where are they located

- "Ju" in ASCII are 74 and 117, 74 + 117 = 191;

  191 % 15 = 11;

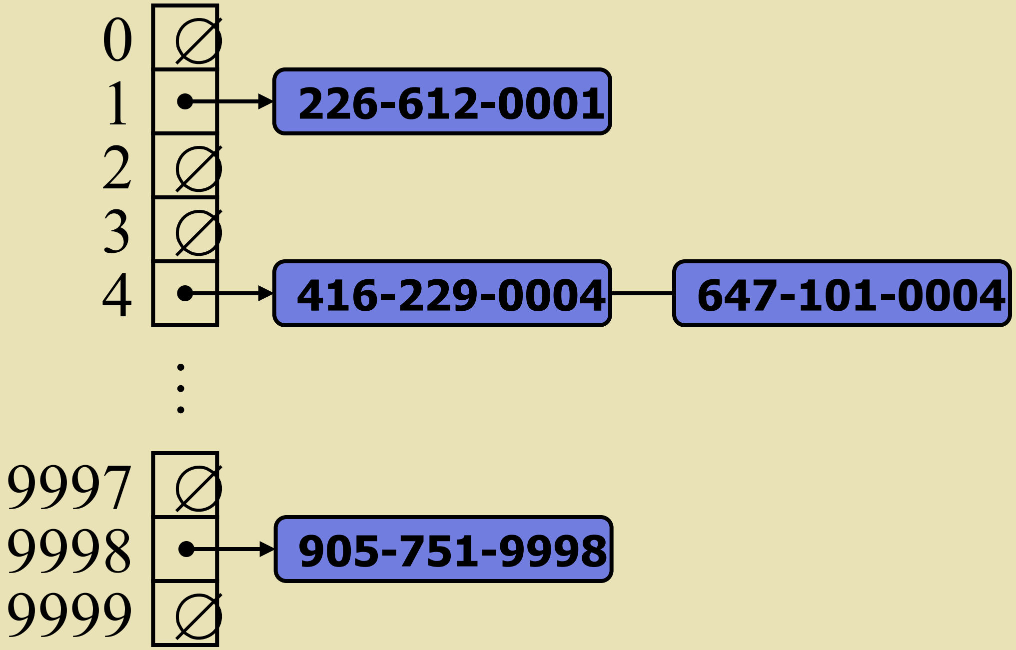
h("Mercury")	= 13
h("Venus")	= 7
h("Earth")	= 1
h("Mars")	= 9
h("Jupiter")	**= 11**
h("Saturn")	= 0
h("Uranus")	= 4
h("Neptune")	= 14
h("Pluto")	= 8

Thus, our search function is simply:

planet search(String s){ return solarSystem[h(s)]; }

Another Example

- We design a hash table for a dictionary storing items (Phone#, Name), where a Phone# is a ten-digit positive integer
- Our hash table uses an array of size $N = 10{,}000$ and the hash function $h(x)$ = last four digits of x
- We use chaining to handle collisions



Hash Functions

- A **hash function** $\boldsymbol{h}()$ maps keys of a given type to integers in a fixed interval $[0, N - 1]$
- Example:

  $$\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{x} \bmod \boldsymbol{N}$$

  is a hash function for integer keys
- The integer returned by $\boldsymbol{h}(\boldsymbol{x})$ is called the **hash value** of key $\boldsymbol{x}$
- The goal of a hash function is to uniformly disperse keys in the range $[0, N - 1]$

Linear Probing for handling collision

- Linear probing handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together. Future collisions may cause a longer sequence of probes

Linear Probing for handling collision

- **Linear probing handles collisions by placing the colliding item in the next (circularly) available table cell**
- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18(5), 41(2), 22(9), 44(5), 59(7), 32(6), 31(5), 73(8), in this order

0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

Empty bucket

- 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.

Inserts using linear probing

- An **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.

Inserts using linear probing

- Algorithm return
  - true
    - if the item was inserted
  - false
    - if all buckets are occupied

- (textbook 6.3.4)

Qtns:

Hash table with linear probing: Insert.

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

1) HashInsert(numsTable, item 13) =3

numsTable:	0	
	1	71
	2	22
	3	13
	4	

Bucket = 3

2) HashInsert(numsTable, item 41) =1x → 2

numsTable:	0	
	1	21
	2	41
	3	
	4	

Bucket = 2

3) HashInsert(numsTable, item 74) =4x → 0x → 1

numsTable:	0	20
	1	74
	2	32
	3	
	4	84

Bucket = 1

Removals using linear probing

- A **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
  Already/ deleted ×
  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. (textbook 6.3.6)

Searching using linear probing

- A **search** algorithm uses the sought item's key to determine the initial bucket. (textbook 6.3.8)
- 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).

Searching using linear probing

- Why the searches algorithm only stops for empty-since-start, not the empty-after-removal?

Searching using linear probing

- Why the searches algorithm only stops for empty-since-start, not the empty-after-removal?

- Item may have been placed in a subsequent bucket before this bucket's item was removed.

Search with Linear Probing

- Consider a hash table *A* that uses linear probing
- findElement(*k*)
  - We start at cell *h*(*k*)
  - We probe consecutive locations until one of the following occurs
    - An item with key *k* is found, or
    - An empty cell is found, or
    - *N* cells have been unsuccessfully probed

```
function findElement(k){
    i = h(k);
    p = 0;
    repeat {
        c = A[i];
        if (c == ∅)
            return NO_SUCH_KEY;
         else if (c.key () == k)
            return c.element()
        else {
            i = (i + 1) mod N;
            p = p + 1;
        } until    ( p == N);
    return NO_SUCH_KEY;
}
```

Quadratic probing

- To avoid collision, **quadratic probing (QP)** starts at the key's mapped bucket, and then quadratically searches subsequent buckets until an empty bucket is found.

$$h(x) = (H + c_1 i + c_2 i^2) \bmod (\text{table size})$$

Hash table insertion using **QP**: $c_1 = 1$ & $c_2 = 1$.

Hash function: key % 10
Quadratic probing sequence: (H + i + i * i) % 10

Operation	H(key)	i	Bucket index	Bucket empty?
Insert key 55	55 % 10 = 5	0	(5 + 0 + 0 * 0) % 10 = 5	Yes
Insert key 66	66 % 10 = 6	0	(6 + 0 + 0 * 0) % 10 = 6	Yes
Insert key 25	25 % 10 = 5	0	(5 + 0 + 0 * 0) % 10 = 5	No
		1	(5 + 1 + 1 * 1) % 10 = 7	Yes

hashTable:

0	
1	
2	
3	
4	
5	55
6	66
7	25
8	
9	

Empty
Occupied

(textbook 6.4.1)

QP

```
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
}
```

Class Exercise

- Assume a hash function returns key % 16 and quadratic probing is used with $c_1 = 1$ & $c_2 = 1$. Refer to the table below. $C + 1 \times i + 1 \times i^2$

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

1) 32 was inserted before 16? True or False?

True

2) Which value was inserted without collision?

32 49 3 23 42 11

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

0 → 2 → 6 → 12

Class Exercise

- Assume a hash function returns key % 16 and quadratic probing is used with $c_1 = 1$ & $c_2 = 1$. Refer to the table below.

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

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

0 -> 2 -> 6 -> ~~12~~

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

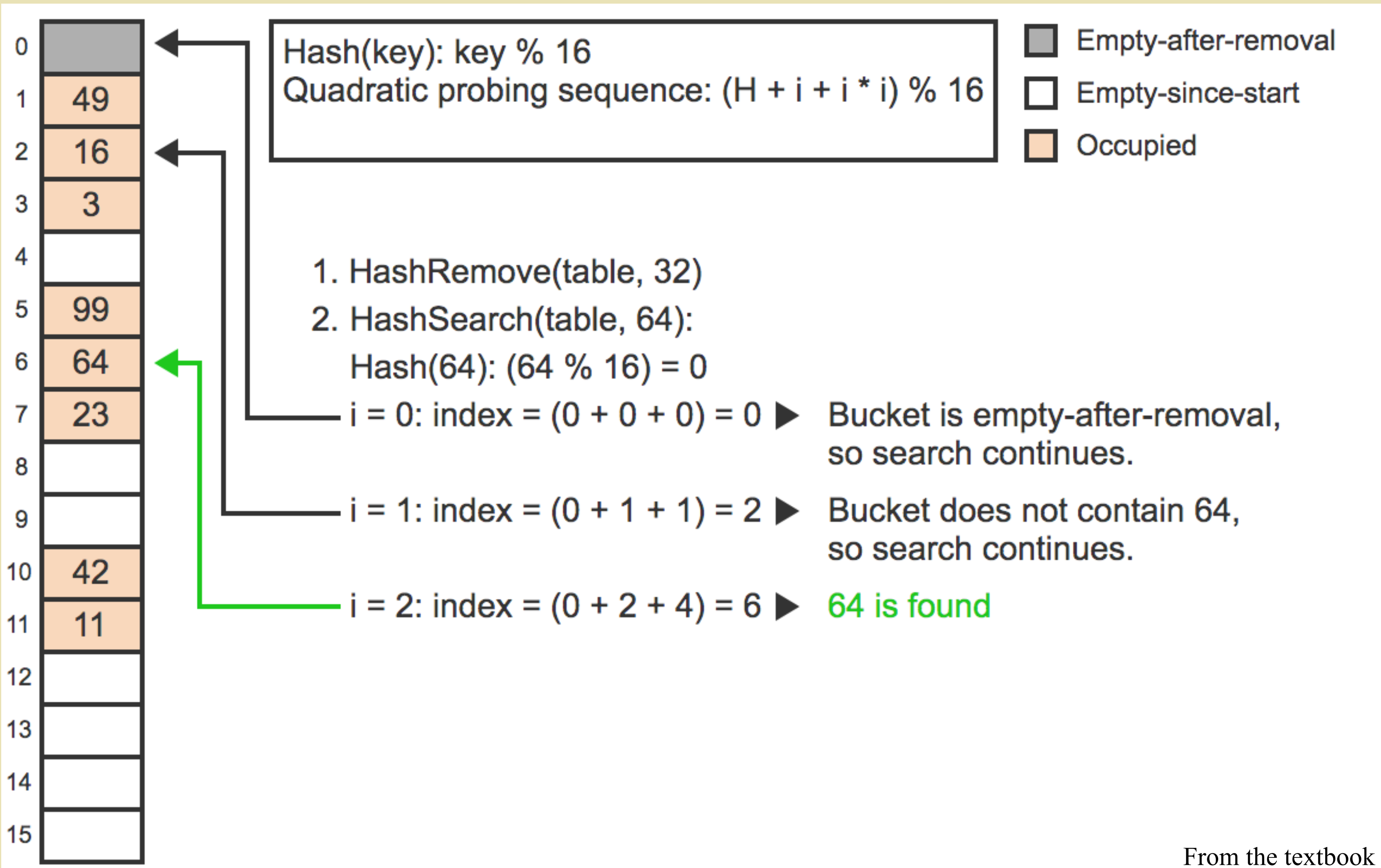
5 -> 7 -> 11 -> 17%16= $\boxed{1}$ -> 20%16= $\boxed{4}$

Questions from the textbook 6.4 and solutions can be found there.

Search & removal

- 6.4.3: Search and removal with quadratic probing: $c_1 = 1$ & $c_2 = 1$.

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



From the textbook 6.4.3.

6.4.4: HT w QP: search and remove

- Consider the following hash table, a hash function of key % 10, and QP with $c_1 = 1$ & $c_2 = 1$:

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

Occupied Empty after deletion Empty

6) HashSearch(valsTable, 75) probes __1__ buckets.?

7) HashSearch(valsTable, 110) probes __2__ buckets.

8) After removing 66 via
HashRemove(valsTable, 66),
HashSearch(valsTable, 66) probes __2__ buckets.

6—5—5

Questions from the textbook 6.4.4 and solutions can be found there.

Double Hashing

Double Hashing for handling collision

- Double hashing uses a secondary hash function $h_2(k)$ and handles collisions by placing an item in the first available cell of the series
  $(h_1(k) + i h_2(k))$ mod N
  for $i = 0, 1, \ldots, N - 1$
- The secondary hash function $h_2(k)$ cannot have zero values
- The table size ***N*** must be a prime to allow probing of all the cells

- Common choice of compression map for the secondary hash function:
  $$h_2(k) = q - k \bmod q$$
  where
  - $q < N$
  - q is a prime
- The possible values for $h_2(k)$ are
  $1, 2, \ldots, q$

Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $N = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - (k \bmod 7)$ 36

  $hash(k) = (h_1(k) + ih_2(k)) \bmod N$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

5 + i×3

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

31		41			18	32	59	73	22	44		
0	1	2	3	4	5	6	7	8	9	10	11	12

Exercises

6.5.2: 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 insertions in order:

16, 77, 55, 41, 63.

0 1 0 10

77 16 6

5

1

0 , 0

5

1) Item 16 Bucket: 5

1) Item 55 Bucket: 10

1) Item 63 Bucket: 2

DH: Insertion, search, and removal

- Read Section 6.5 of text

Performance of Probing:

- Let **N** be the number of slots of a hash table, **n** be the number of items in the table, we define load factor as:

  $$\alpha = n/N$$

- If the hash function randomly distributes keys through the table, then the expected length of a successful search path is:

  $$\text{length}_{succ} = ½\,(1 + 1/(1- \alpha))$$

  $$\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

Performance of Probing:

- The expected length of an unsuccessful search is approximately:

$$\text{length}_{\text{unsucc}} = \tfrac{1}{2}\left(1 + 1/(1 - \alpha)^2\right)$$

That's about this lecture!