DATA STRUCTURES AND ALGORITHMS

LECTURE 9

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2019 - 2020

In Lecture 8...

- Binary heap
- Binomial heap
- Hash tables
  - Separate chaining

Today

- Hash tables
  - Separate chaining
  - Hash function
  - Coalesced chaining
  - Open addressing

Separate chaining

- Collision resolution by chaining: each slot from the hash table T contains a linked list, with the elements that hash to that slot

Node:
key: TKey
next: ↑ Node

HashTable:
T: ↑Node[] *//an array of pointers to nodes*
m: Integer
h: TFunction *//the hash function*

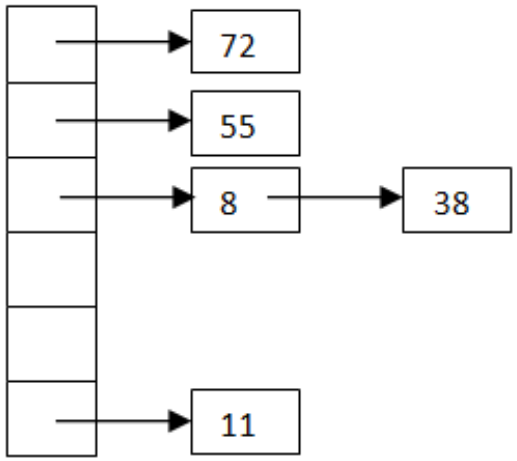
Example

- Assume we have a hash table with m = 6 that uses separate chaining for collision resolution, with the following policy: if the load factor of the table after an insertion is greater than or equal to 0.7, we double the size of the table.

- Using the division method for the hash function, insert the following elements, in the given order, in the hash table: 38, 11, 8, 72, 57, 29, 2.

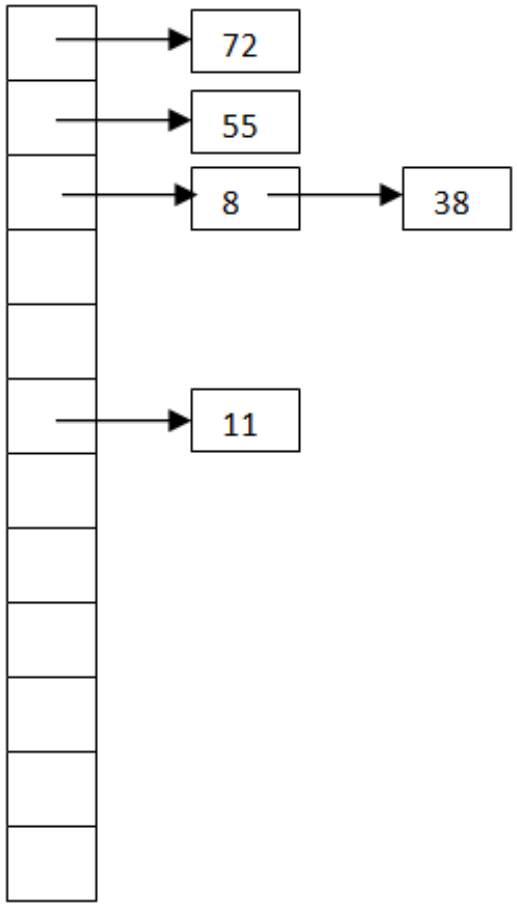
Example

- $h(38) = 2$ (load factor will be $1/6$)
- $h(11) = 5$ (load factor will be $2/6$)
- $h(8) = 2$ (load factor will be $3/6$)
- $h(72) = 0$ (load factor will be $4/6$)
- $h(55) = 1$ (load factor will be $5/6$ - greater than 0.7)
- The table after the first five elements were added:



Example

- Is it OK if after the resize this is our hash table?

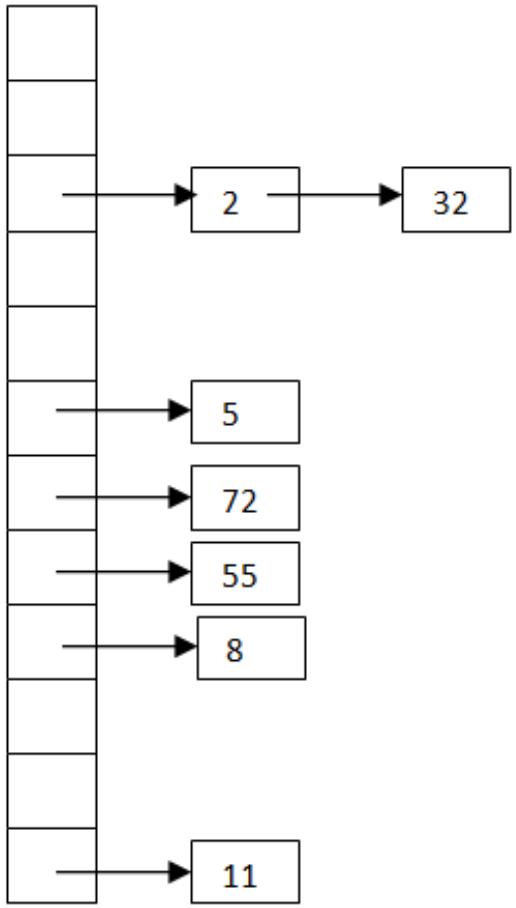


Example

- The result of the hash function (i.e. the position where an element is added) depends on the size of the hash table. If the size of the hash table changes, the value of the hash function changes as well, which means that search and remove operations might not find the element.
- After a resize operation, we have to add all elements again in the hash table, to make sure that they are at the correct position $\rightarrow$ rehash

Example

- After rehash and adding the other two elements:



Iterator

- What do you think, which containers cannot be represented on a hash table?

Iterator

- What do you think, which containers cannot be represented on a hash table?
- How can we define an iterator for a hash table with separate chaining?

Iterator

- What do you think, which containers cannot be represented on a hash table?
- How can we define an iterator for a hash table with separate chaining?
- Since hash tables are used to implement containers where the order of the elements is not important, our iterator can iterate through them in any order.
- For the hash table from the previous example, the easiest order in which the elements can be iterated is: 2, 32, 5, 72, 55, 8, 11

Iterator

- Iterator for a hash table with separate chaining is a combination of an iterator on an array (table) and on a linked list.
- We need a current position to know the position from the table that we are at, but we also need a current node to know the exact node from the linked list from that position.

IteratorHT:
ht: HashTable
currentPos: Integer
currentNode: ↑ Node

Iterator - init

- How can we implement the *init* operation?

Iterator - init

- How can we implement the *init* operation?

```
subalgorithm init(ith, ht) is:
//pre: ith is an IteratorHT, ht is a HashTable
    ith.ht ← ht
    ith.currentPos ← 0
    while ith.currentPos < ht.m and ht.T[ith.currentPos] = NIL execute
        ith.currentPos ← ith.currentPos + 1
    end-while
    if ith.currentPos < ht.m then
        ith.currentNode ← ht.T[ith.currentPos]
    else
        ith.currentNode ← NIL
    end-if
end-subalgorithm
```

- Complexity of the algorithm:

Iterator - init

- How can we implement the *init* operation?

```
subalgorithm init(ith, ht) is:
//pre: ith is an IteratorHT, ht is a HashTable
    ith.ht ← ht
    ith.currentPos ← 0
    while ith.currentPos < ht.m and ht.T[ith.currentPos] = NIL execute
        ith.currentPos ← ith.currentPos + 1
    end-while
    if ith.currentPos < ht.m then
        ith.currentNode ← ht.T[ith.currentPos]
    else
        ith.currentNode ← NIL
    end-if
end-subalgorithm
```

- Complexity of the algorithm: $O(m)$

Iterator - other operations

- How can we implement the *getCurrent* operation?

Iterator - other operations

- How can we implement the *getCurrent* operation?
- How can we implement the *next* operation?

Iterator - other operations

- How can we implement the *getCurrent* operation?
- How can we implement the *next* operation?
- How can we implement the *valid* operation?

The hash function

- The main goal of the hash function is to map an element to a position in the hash table.

$$h : U \to \{0, 1, ..., m-1\}$$

A good hash function

- A good hash function:
  - can minimize the number of collisions (but cannot eliminate all collisions)
  - is deterministic
  - can be computed in $\Theta(1)$ time
  - is computed using all parts of the data
  - satisfies (approximately) the assumption of simple uniform hashing: **each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to**

$$P(h(k) = j) = \frac{1}{m} \ \forall j = 0, ..., m-1 \ \ \forall k \in U$$

Examples of bad hash functions

- $h(k) =$ constant number

Examples of bad hash functions

- $h(k) =$ constant number
- $h(k) =$ random number

Examples of bad hash functions

- $h(k)$ = constant number
- $h(k)$ = random number
- assuming that the keys are CNP numbers:
  - a hash function considering just parts of it (first digit, birth year/date, county code, etc.)
  - assume m = 100 and you use the birth day from the CNP (as a number): h(CNP) = birthday % 100

Examples of bad hash functions

- $h(k)$ = constant number
- $h(k)$ = random number
- assuming that the keys are CNP numbers:
  - a hash function considering just parts of it (first digit, birth year/date, county code, etc.)
  - assume m = 100 and you use the birth day from the CNP (as a number): h(CNP) = birthday % 100
- m = 16 and h(k) % m can also be problematic

Examples of bad hash functions

- $h(k)$ = constant number
- $h(k)$ = random number
- assuming that the keys are CNP numbers:
  - a hash function considering just parts of it (first digit, birth year/date, county code, etc.)
  - assume m = 100 and you use the birth day from the CNP (as a number): h(CNP) = birthday % 100
- m = 16 and h(k) % m can also be problematic
- etc.

Hash function

- The simple uniform hashing theorem is hard to satisfy, especially when we do not know the distribution of data. Data does not always have a uniform distribution
  - dates
  - group numbers at our faculty
  - postal codes
  - first letter of an English word
- In practice we use heuristic techniques to create hash functions that perform well.

- Most hash functions assume that the keys are natural numbers. If this is not true, they have to be interpreted as natural number. In what follows, we assume that the keys are natural numbers.

The division method

The division method

$h(k) = k \mod m$

For example:

$$m = 13$$
$$k = 63 => h(k) = 11$$
$$k = 52 => h(k) = 0$$
$$k = 131 => h(k) = 1$$

- Requires only a division so it is quite fast
- Experiments show that good values for m are primes not too close to exact powers of 2

The division method

- Interestingly, Java uses the division method with a table size which is power of 2 (initially 16).
- They avoid a problem, by using a second function for hashing, before applying the mod:

```
/**
 * Applies a supplemental hash function to a given hashCode, which
 * defends against poor quality hash functions.  This is critical
 * because HashMap uses power-of-two length hash tables, that
 * otherwise encounter collisions for hashCodes that do not differ
 * in lower bits. Note: Null keys always map to hash 0, thus index 0.
 */
static int hash(int h) {
    // This function ensures that hashCodes that differ only by
    // constant multiples at each bit position have a bounded
    // number of collisions (approximately 8 at default load factor).
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
```

Mid-square method

- Assume that the table size is 10^r, for example m = 100 (r = 2)
- For getting the hash of a number, multiply it by itself and take the middle r digits.
- For example, h(4567) = middle 2 digits of 4567 * 4567 = middle 2 digits of 20857489 = 57
- Same thing works for $m = 2^r$ and the binary representation of the numbers
- $m = 2^4$, h(1011) = middle 4 digits of 01111001 = 1110

The multiplication method I

The multiplication method

$h(k) = floor(m * frac(k * A))$ where

m - the hash table size

A - constant in the range $0 < A < 1$

$frac(k * A)$ - fractional part of $k * A$

For example

m = 13 A = 0.6180339887

k=63 => h(k) = floor(13 * frac(63 * A)) = floor(12.16984) = 12

k=52 => h(k) = floor(13 * frac(52 * A)) = floor(1.790976) = 1

k=129=> h(k)= floor(13 * frac(129 * A)) = floor(9.442999) = 9

The multiplication method II

- Advantage: the value of m is not critical, typically $m = 2^p$ for some integer p
- Some values for A work better than others. Knuth suggests $\frac{\sqrt{5}-1}{2} = 0.6180339887$

Universal hashing I

- If we know the exact hash function used by a hash table, we can always generate a set of keys that will hash to the same position (collision). This reduces the performance of the table.
- For example:

$m = 13$
$h(k) = k \ \ mod \ \ m$
k = 11, 24, 37, 50, 63, 76, etc.

Universal hashing II

- Instead of having one hash function, we have a collection $\mathcal{H}$ of hash functions that map a given universe U of keys into the range $\{0, 1, \ldots, m-1\}$
- Such a collection is said to be **universal** if for each pair of distinct keys $x, y \in U$ the number of hash functions from $\mathcal{H}$ for which $h(x) = h(y)$ is precisely $\frac{|\mathcal{H}|}{m}$
- In other words, with a hash function randomly chosen from $\mathcal{H}$ the chance of collision between x and y, where $x \neq y$, is exactly $\frac{1}{m}$

Universal hashing III

Example 1

Fix a prime number $p >$ *the maximum possible value for a key from* U.
For every $a \in \{1, \ldots, p-1\}$ and $b \in \{0, \ldots, p-1\}$ we can define a hash function $h_{a,b}(k) = ((a * k + b) \ mod \ p) \ mod \ m$.

- For example:
  - $h_{3,7}(k) = ((3 * k + 7) \ mod \ p) \ mod \ m$
  - $h_{4,1}(k) = ((4 * k + 1) \ mod \ p) \ mod \ m$
  - $h_{8,0}(k) = ((8 * k) \ mod \ p) \ mod \ m$
- There are $p * (p-1)$ possible hash functions that can be chosen.

Universal hashing IV

Example 2

If the key k is an array $< k_1, k_2, \ldots, k_r >$ such that $k_i < m$ (or it can be transformed into such an array, by writing the k as a number in base m).
Let $< x_1, x_2, \ldots, x_r >$ be a fixed sequence of random numbers, such that $x_i \in \{0, \ldots, m-1\}$ (another number in base m with the same length).
$h(k) = \sum_{i=1}^{r} k_i * x_i \mod m$

Example 3

Suppose the keys are $u - bits$ long and $m = 2^b$.
Pick a random $b - by - u$ matrix (called h) with 0 and 1 values only.
Pick $h(k) = h * k$ where in the multiplication we do addition mod 2.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$

Using keys that are not natural numbers I

- The previously presented hash functions assume that keys are natural numbers.
- If this is not true there are two options:
  - Define special hash functions that work with your keys (for example, for real number from the [0,1) interval $h(k) = [k * m]$ can be used)
  - Use a function that transforms the key to a natural number (and use any of the above-mentioned hash functions) - *hashCode* in Java, *hash* in Python

Using keys that are not natural numbers II

- If the key is a string s:
  - we can consider the ASCII codes for every letter
  - we can use 1 for *a*, 2 for *b*, etc.
- Possible implementations for *hashCode*
  - $s[0] + s[1] + ... + s[n-1]$
    - Anagrams have the same sum *SAUCE* and *CAUSE*
    - *DATES* has the same sum (D = C + 1, T = U - 1)
    - Assuming maximum length of 10 for a word (and the second letter representation), *hashCode* values range from 1 (the word *a*) to 260 (*zzzzzzzzzz*). Considering a dictionary of about 50,000 words, we would have on average 192 word for a *hashCode* value.

Using keys that are not natural numbers III

- $s[0] * 26^{n-1} + s[1] * 26^{n-2} + ... + s[n-1]$ where
  - n - the length of the string
  - Generates a much larger interval of *hashCode* values.
  - Instead of 26 (which was chosen since we have 26 letters) we can use a prime number as well (Java uses 31, for example).

Cryptographic hashing

- Another use of hash functions besides as part of a hash table
- It is a hash function, which can be used to generate a code (the hash value) for any variable size data
- Used for checksums, storing passwords, etc.

Coalesced chaining

- Collision resolution by coalesced chaining: each element from the hash table is stored inside the table (no linked lists), but each element has a *next* field, similar to a linked list on array.
- When a new element has to be inserted and the position where it should be placed is occupied, we will put it to any empty position, and set the *next* link, so that the element can be found in a search.
- Since elements are in the table, α can be at most 1.

Coalesced chaining - example

- Consider a hash table of size $m = 16$ that uses coalesced chaining for collision resolution and a hash function with the division method
- Insert into the table the following elements: 76, 12, 109, 43, 22, 18, 55, 81, 91, 27, 13, 16, 39.
- Let's compute the value of the hash function for every key:

Key	76	12	109	43	22	18	55	81	91	27	13	16	39
Hash	12	12	13	11	6	2	7	1	11	11	13	0	7

Example

- Initially the hash table is empty. All next values are -1 and the first empty position is position 0.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

firstEmpty = 0

- 76 will be added to position 12. But 12 should also be added there. Since that position is already occupied, we add 12 to position firstEmpty and set the next of 76 to point to position 0. Then we reset firstEmpty to the next empty position

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12												76			
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	-1	-1	-1

firstEmpty = 1

Example

- And we continue in the same manner. We have no collisions up to 81, but we need to reset firstEmpty when we *accidentally* occupy it.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12	81	18				22	55				43	76	109		
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	0	-1	-1	-1

firstEmpty = 3

- When adding 91, we put it to position firstEmpty and set the next link of position 11 to position 3.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12	81	18	91			22	55				43	76	109		
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	3	0	-1	-1	-1

firstEmpty = 4

Example

- The final table:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12	81	18	91	27	13	22	55	16	39		43	76	109		
8	-1	-1	4	-1	-1	-1	9	-1	-1	-1	3	0	5	-1	-1

firstEmpty = 10

Coalesced chaining - representation

- What fields do we need to represent a hash table where collision resolution is done with coalesced chaining?

Coalesced chaining - representation

- What fields do we need to represent a hash table where collision resolution is done with coalesced chaining?

HashTable:
T: TKey[]
next: Integer[]
m: Integer
firstEmpty: Integer
h: TFunction

- For simplicity, in the following, we will consider only the keys.

Coalesced chaining - insert

```
subalgorithm insert (ht, k) is:
//pre: ht is a HashTable, k is a TKey
//post: k was added into ht
    pos ← ht.h(k)
    if ht.T[pos] = -1 then //-1 means empty position
        ht.T[pos] ← k
        ht.next[pos] ← -1
    else
        if ht.firstEmpty = ht.m then
            @resize and rehash
        end-if
        current ← pos
        while ht.next[current] ≠ -1 execute
            current ← ht.next[current]
        end-while
//continued on the next slide...
```

Coalesced chaining - insert

```
        ht.T[ht.firstEmpty] ← k
        ht.next[ht.firstEmpty] ← - 1
        ht.next[current] ← ht.firstEmpty
        changeFirstEmpty(ht)
    end-if
end-subalgorithm
```

- Complexity: $\Theta(1)$ on average, $\Theta(n)$ - worst case

Coalesced chaining - ChangeFirstEmpty

subalgorithm changeFirstEmpty(ht) **is:**
//pre: ht is a HashTable
//post: the value of ht.firstEmpty is set to the next free position
  ht.firstEmpty $\leftarrow$ ht.firstEmpty + 1
  while ht.firstEmpty $<$ ht.m **and** ht.T[ht.firstEmpty] $\neq$ -1
execute
    ht.firstEmpty $\leftarrow$ ht.firstEmpty + 1
  end-while
end-subalgorithm

- Complexity: $O(m)$
- *Think about it:* Should we keep the free spaces linked in a list as in case of a linked lists on array?

Coalesced chaining

- *Remove* and *search* operations for coalesced chaining will be discussed in Seminar 6.
- How can we define an iterator for a hash table with coalesced chaining? What should the following operations do?
  - init
  - getCurrent
  - next
  - valid
- How can we implement a sorted container on a hash table with coalesced chaining? How can we implement its iterator?

Open addressing

- In case of open addressing every element of the hash table is inside the table, we have no pointers, no next links.
- When we want to insert a new element, we will successively generate positions for the element, check (*probe*) the generated position, and place the element in the first available one.

Open addressing

- In order to generate multiple positions, we will extend the hash function and add to it another parameter, i, which is the *probe number* and starts from 0.

$$h : U \times \{0, 1, ..., m-1\} \rightarrow \{0, 1, ..., m-1\}$$

- For an element k, we will successively examine the positions $< h(k, 0), h(k, 1), h(k, 2), ..., h(k, m-1) >$ - called the *probe sequence*
- The *probe sequence* should be a permutation of a hash table positions $\{0, ..., m-1\}$, so that eventually every slot is considered.
- We would also like to have a hash function which can generate all the m! permutations possible (spoiler alert: we cannot)

Open addressing - Linear probing

- One version of defining the hash function is to use linear probing:

$$h(k,i) = (h'(k) + i) \ mod \ m \ \ \forall i = 0, ..., m-1$$

- where $h'(k)$ is a *simple* hash function (for example: $h'(k) = k \ mod \ m$)

- the *probe sequence* for linear probing is: $< h'(k), h'(k)+1, h'(k)+2, ..., m-1, 0, 1, ..., h'(k)-1 >$

Open addressing - Linear probing - example

- Consider a hash table of size $m = 16$ that uses open addressing and linear probing for collision resolution
- Insert into the table the following elements: 76, 12, 109, 43, 22, 18, 55, 81, 91, 27, 13, 16, 39.
- Let's compute the value of the hash function for every key for i = 0:

Key	76	12	109	43	22	18	55	81	91	27	13	16	39
Hash	12	12	13	11	6	2	7	1	11	11	13	0	7

Open addressing - Linear probing - example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
27	81	18	13	16		22	55	39			43	76	12	109	91

Open addressing - Linear probing - example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
27	81	18	13	16		22	55	39			43	76	12	109	91

- Disadvantages of linear probing:
  - There are only m distinct probe sequences (once you have the starting position everything is fixed)
  - *Primary clustering* - long runs of occupied slots
- Advantages of linear probing:
  - Probe sequence is always a permutation
  - Can benefit from caching

Open addressing - Linear probing - primary clustering

- Why is primary clustering a problem?
- Assume m positions, n elements and $\alpha = 0.5$ (so $n = m/2$)
- Best case arrangement: every second position is empty (for example: even positions are occupied and odd ones are free)
- What is the average number probes (positions verified) that need to be checked to insert a new element?

Open addressing - Linear probing - primary clustering

- Why is primary clustering a problem?
- Assume m positions, n elements and $\alpha = 0.5$ (so $n = m/2$)
- Best case arrangement: every second position is empty (for example: even positions are occupied and odd ones are free)
- What is the average number probes (positions verified) that need to be checked to insert a new element?

- Worst case arrangement: all n elements are one after the other (assume in the second half of the array)
- What is the average number of probes (positions verified) that need to be checked to insert a new element?

Open addressing - Quadratic probing

- In case of quadratic probing the hash function becomes:

$$h(k,i) = (h'(k) + c_1 * i + c_2 * i^2) \ mod \ m \ \ \forall i = 0, ..., m-1$$

- where $h'(k)$ is a *simple* hash function (for example: $h'(k) = k \ mod \ m$) and c_1 and c_2 are constants initialized when the hash function is initialized. c_2 should not be 0.

- Considering a simplified version of $h(k,i)$ with $c_1 = 0$ and $c_2 = 1$ the probe sequence would be: $< k, k+1, k+4, k+9, k+16, ... >$

Open addressing - Quadratic probing

- One important issue with quadratic probing is how we can choose the values of m, c_1 and c_2 so that the probe sequence is a permutation.
- If m is a prime number only the first half of the probe sequence is unique, so, once the hash table is half full, there is no guarantee that an empty space will be found.
  - For example, for $m = 17$, $c_1 = 3$, $c_2 = 1$ and $k = 13$, the probe sequence is $< 13, 0, 6, 14, 7, 2, 16, 15, 16, 2, 7, 14, 6, 0, 13, 11, 11 >$
  - For example, for $m = 11$, $c_1 = 1$, $c_2 = 1$ and $k = 27$, the probe sequence is $< 5, 7, 0, 6, 3, 2, 3, 6, 0, 7, 5 >$

Open addressing - Quadratic probing

- If m is a power of 2 and $c_1 = c_2 = 0.5$, the probe sequence will always be a permutation. For example for $m = 8$ and $k = 3$:
  - $h(3, 0) = (3 \,\%\, 8 + 0.5 * 0 + 0.5 * 0^2) \,\%\, 8 = 3$
  - $h(3, 1) = (3 \,\%\, 8 + 0.5 * 1 + 0.5 * 1^2) \,\%\, 8 = 4$
  - $h(3, 2) = (3 \,\%\, 8 + 0.5 * 2 + 0.5 * 2^2) \,\%\, 8 = 6$
  - $h(3, 3) = (3 \,\%\, 8 + 0.5 * 3 + 0.5 * 3^2) \,\%\, 8 = 1$
  - $h(3, 4) = (3 \,\%\, 8 + 0.5 * 4 + 0.5 * 4^2) \,\%\, 8 = 5$
  - $h(3, 5) = (3 \,\%\, 8 + 0.5 * 5 + 0.5 * 5^2) \,\%\, 8 = 2$
  - $h(3, 6) = (3 \,\%\, 8 + 0.5 * 6 + 0.5 * 6^2) \,\%\, 8 = 0$
  - $h(3, 7) = (3 \,\%\, 8 + 0.5 * 7 + 0.5 * 7^2) \,\%\, 8 = 7$

Open addressing - Quadratic probing

- If m is a prime number of the form $4 * k + 3$, $c_1 = 0$ and $c_2 = (-1)^i$ (so the probe sequence is +0, -1, +4, -9, etc.) the probe sequence is a permutation. For example for $m = 7$ and $k = 3$:
  - $h(3, 0) = (3 \,\%\, 7 + 0^2) \,\%\, 7 = 3$
  - $h(3, 1) = (3 \,\%\, 7 - 1^2) \,\%\, 7 = 2$
  - $h(3, 2) = (3 \,\%\, 7 + 2^2) \,\%\, 7 = 0$
  - $h(3, 3) = (3 \,\%\, 7 - 3^2) \,\%\, 7 = 1$
  - $h(3, 4) = (3 \,\%\, 7 + 4^2) \,\%\, 7 = 5$
  - $h(3, 5) = (3 \,\%\, 7 - 5^2) \,\%\, 7 = 6$
  - $h(3, 6) = (3 \,\%\, 7 + 6^2) \,\%\, 7 = 4$

Open addressing - Quadratic probing - example

- Consider a hash table of size $m = 16$ that uses open addressing with quadratic probing for collision resolution (h'(k) is a hash function defined with the division method), $c_1 = c_2 = 0.5$.
- Insert into the table the following elements: 76, 12, 109, 43, 22, 18, 55, 81, 91, 27, 13, 16, 39.

Open addressing - Quadratic probing - example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
13	81	18	16		91	22	55	39		27	43	76	12	109	

Open addressing - Quadratic probing - example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
13	81	18	16		91	22	55	39		27	43	76	12	109	

- Disadvantages of quadratic probing:
  - The performance is sensitive to the values of m, c_1 and c_2.
  - *Secondary clustering* - if two elements have the same initial probe positions, their whole probe sequence will be identical: $h(k_1, 0) = h(k_2, 0) \Rightarrow h(k_1, i) = h(k_2, i)$.
  - There are only m distinct probe sequences (once you have the starting position the whole sequence is fixed).