Collision Resolution: Open Addressing

- Quadratic Probing
- Double Hashing
- Rehashing
- Algorithms for:
  - insert
  - find
  - withdraw

Open Addressing: Quadratic Probing

- Quadratic probing eliminates primary clusters.
- **c(i)** is a quadratic function in **i** of the form **c(i) = a*i^2 + b*i**. Usually **c(i)** is chosen as:

  $c(i) = i^2$ **for i = 0, 1, . . . , tableSize – 1**

  or

  $c(i) = \pm i^2$ **for i = 0, 1, . . . , (tableSize – 1) / 2**

- The probe sequences are then given by:

  $h_i(key) = [h(key) + i^2]$ % tableSize **for i = 0, 1, . . . , tableSize – 1**

  or

  $h_i(key) = [h(key) \pm i^2]$ % tableSize **for i = 0, 1, . . . , (tableSize – 1) / 2**

- Note for Quadratic Probing:
  - Hashtable size should not be an even number; otherwise Property 2 will not be satisfied.
  - Ideally, table size should be a prime of the form 4j+3, where j is an integer. This choice of table size guarantees Property 2.

Quadratic Probing (cont'd)

- Example: Load the keys **23, 13, 21, 14, 7, 8, and 15**, in this order, in a hash table of size **7** using quadratic probing with **c(i) = ±i²** and the hash function: **h(key) = key % 7**
- The required probe sequences are given by:

$$h_i(key) = (h(key) \pm i^2) \% 7 \quad i = 0, 1, 2, 3$$

Quadratic Probing (cont'd)

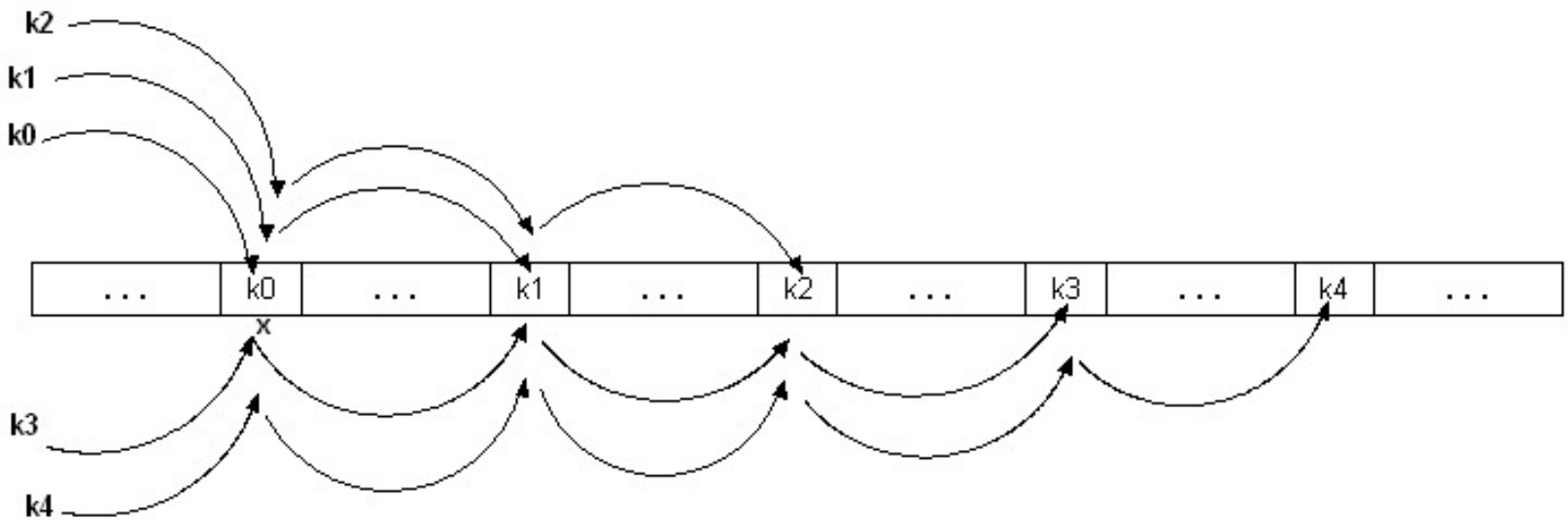
$h_i(key) = (h(key) \pm i^2) \% 7 \quad i = 0, 1, 2, 3$

$h_0(23) = (23 \% 7) \% 7 = 2$
$h_0(13) = (13 \% 7) \% 7 = 6$
$h_0(21) = (21 \% 7) \% 7 = 0$
$h_0(14) = (14 \% 7) \% 7 = 0$ collision
$h_1(14) = (0 + 1^2) \% 7 = 1$
$h_0(7) = (7 \% 7) \% 7 = 0$ collision
$h_1(7) = (0 + 1^2) \% 7 = 1$ collision
$h_{-1}(7) = (0 - 1^2) \% 7 = -1$
NORMALIZE: $(-1 + 7) \% 7 = 6$ collision
$h_2(7) = (0 + 2^2) \% 7 = 4$
$h_0(8) = (8 \% 7) \% 7 = 1$ collision
$h_1(8) = (1 + 1^2) \% 7 = 2$ collision
$h_{-1}(8) = (1 - 1^2) \% 7 = 0$ collision
$h_2(8) = (1 + 2^2) \% 7 = 5$
$h_0(15) = (15 \% 7) \% 7 = 1$ collision
$h_1(15) = (1 + 1^2) \% 7 = 2$ collision
$h_{-1}(15) = (1 - 1^2) \% 7 = 0$ collision
$h_2(15) = (1 + 2^2) \% 7 = 5$ collision
$h_{-2}(15) = (1 - 2^2) \% 7 = -3$
NORMALIZE: $(-3 + 7) \% 7 = 4$ collision
$h_3(15) = (1 + 3^2) \% 7 = 3$

0	O	21
1	O	14
2	O	23
3	O	15
4	O	7
5	O	8
6	O	13

Secondary Clusters

- Quadratic probing is better than linear probing because it eliminates primary clustering.
- However, it may result in **secondary clustering**: if **h(k1) = h(k2)** the probing sequences for **k1** and **k2** are exactly the same. This sequence of locations is called a secondary cluster.
- Secondary clustering is less harmful than primary clustering because secondary clusters do not combine to form large clusters.
- **Example of Secondary Clustering:** Suppose keys **k0, k1, k2, k3, and k4** are inserted in the given order in an originally empty hash table using **quadratic probing** with **c(i) = i²**. Assuming that each of the keys hashes to the same array index **x**. A secondary cluster will develop and grow in size:



Double Hashing

- To eliminate secondary clustering, synonyms must have different probe sequences.
- Double hashing achieves this by having two hash functions that both depend on the hash key.
- $c(i) = i * h_p(key)$ **for i = 0, 1, . . . , tableSize – 1**
  where h_p (or h_2) is another hash function.
- The probing sequence is:
  $h_i(key) = [h(key) + i*h_p(key)]\%$ tableSize **for i = 0, 1, . . . , tableSize – 1**
- The function $c(i) = i*h_p(r)$ satisfies Property 2 provided $h_p(r)$ and tableSize are relatively prime.
- To guarantee Property 2, tableSize must be a prime number.
- Common definitions for $\mathbf{h_p}$ are :
  - $h_p(key) = 1 + key \% (tableSize - 1)$
  - $h_p(key) = q - (key \% q)$ where **q** is a prime less than **tableSize**
  - $h_p(key) = q*(key \% q)$ where **q** is a prime less than **tableSize**

Double Hashing (cont'd)

Performance of Double hashing:

- Much better than linear or quadratic probing because it eliminates both primary and secondary clustering.
- BUT requires a computation of a second hash function $\mathbf{h_p}$.

Example: Load the keys **18, 26, 35, 9, 64, 47, 96, 36, and 70** in this order, in an empty hash table of size **13**

(a) using double hashing with the first hash function: **h(key) = key % 13** and the second hash function: $\mathbf{h_p(key) = 1 + key \% 12}$

(b) using double hashing with the first hash function: **h(key) = key % 13** and the second hash function: $\mathbf{h_p(key) = 7 - key \% 7}$

Show all computations.

Double Hashing (cont'd)

0	1	2	3	4	5	6	7	8	9	10	11	12
26			70		18	9	96	47	35	36		64

$h_i(key) = [h(key) + i*h_p(key)]\% 13$

$h(key) = key \% 13$

$h_p(key) = 1 + key \% 12$

$h_0(18) = (18\%13)\%13 = 5$

$h_0(26) = (26\%13)\%13 = 0$

$h_0(35) = (35\%13)\%13 = 9$

$h_0(9) = (9\%13)\%13 = 9$ collision

$h_p(9) = 1 + 9\%12 = 10$

$h_1(9) = (9 + 1*10)\%13 = 6$

$h_0(64) = (64\%13)\%13 = 12$

$h_0(47) = (47\%13)\%13 = 8$

$h_0(96) = (96\%13)\%13 = 5$ collision

$h_p(96) = 1 + 96\%12 = 1$

$h_1(96) = (5 + 1*1)\%13 = 6$ collision

$h_2(96) = (5 + 2*1)\%13 = 7$

$h_0(36) = (36\%13)\%13 = 10$

$h_0(70) = (70\%13)\%13 = 5$ collision

$h_p(70) = 1 + 70\%12 = 11$

$h_1(70) = (5 + 1*11)\%13 = 3$

Double Hashing (cont'd)

0	1	2	3	4	5	6	7	8	9	10	11	12
26	9				18	70	96	47	35	36		64

$h_0(18) = (18\%13)\%13 = 5$
$h_0(26) = (26\%13)\%13 = 0$
$h_0(35) = (35\%13)\%13 = 9$
$h_0(9) = (9\%13)\%13 = 9$ collision
$h_p(9) = 7 - 9\%7 = 5$
$h_1(9) = (9 + 1*5)\%13 = 1$
$h_0(64) = (64\%13)\%13 = 12$
$h_0(47) = (47\%13)\%13 = 8$
$h_0(96) = (96\%13)\%13 = 5$ collision
$h_p(96) = 7 - 96\%7 = 2$
$h_1(96) = (5 + 1*2)\%13 = 7$
$h_0(36) = (36\%13)\%13 = 10$
$h_0(70) = (70\%13)\%13 = 5$ collision
$h_p(70) = 7 - 70\%7 = 7$
$h_1(70) = (5 + 1*7)\%13 = 12$ collision
$h_2(70) = (5 + 2*7)\%13 = 6$

$h_i(key) = [h(key) + i*h_p(key)]\% 13$

$h(key) = key \% 13$

$h_p(key) = 7 - key \% 7$

Rehashing

- As noted before, with open addressing, if the hash tables become too full, performance can suffer a lot.
- So, what can we do?
- We can double the hash table size, modify the hash function, and re-insert the data.
  - More specifically, the new size of the table will be the first prime that is more than twice as large as the old table size.

Implementation of Open Addressing

```
public class OpenScatterTable extends AbstractHashTable {
   protected Entry array[];
   protected static final int EMPTY = 0;
   protected static final int OCCUPIED = 1;
   protected static final int DELETED = 2;

   protected static final class Entry {
      public int state = EMPTY;
      public Comparable object;
      // …
   }

   public OpenScatterTable(int size) {
      array = new Entry[size];
      for(int i = 0; i < size; i++)
         array[i] = new Entry();
   }
   // …
}
```

Implementation of Open Addressing (Con't.)

```
/* finds the index of the first unoccupied slot
   in the probe sequence of obj */
protected int findIndexUnoccupied(Comparable obj){
   int hashValue = h(obj);
   int tableSize = getLength();
   int indexDeleted = -1;
   for(int i = 0; i < tableSize; i++){
      int index = (hashValue + c(i)) % tableSize;
      if(array[index].state == OCCUPIED
                && obj.equals(array[index].object))
         throw new IllegalArgumentException(
                          "Error: Duplicate key");

      else if(array[index].state == EMPTY ||
        (array[index].state == DELETED &&
            obj.equals(array[index].object)))
        return indexDeleted ==-1?index:indexDeleted;
      else if(array[index].state == DELETED &&
        indexDeleted == -1)
          indexDeleted = index;
   }
   if(indexDeleted != -1) return indexDeleted;

   throw new IllegalArgumentException(
                "Error: Hash table is full");
}
```

Implementation of Open Addressing (Con't.)

```
protected int findObjectIndex(Comparable obj){
   int hashValue = h(obj);
   int tableSize = getLength();

   for(int i = 0; i < tableSize; i++){
      int index = (hashValue + c(i)) % tableSize;
      if(array[index].state == EMPTY
            || (array[index].state == DELETED
            && obj.equals(array[index].object)))
         return -1;
      else if(array[index].state == OCCUPIED
               && obj.equals(array[index].object))
         return index;
   }
   return -1;
}

public Comparable find(Comparable obj){
   int index = findObjectIndex(obj);
   if(index >= 0)return array[index].object;
   else return null;
}
```

Implementation of Open Addressing (Con't.)

```
public void insert(Comparable obj){
   if(count == getLength()) throw new ContainerFullException();
   else {
      int index = findIndexUnoccupied(obj);
      // throws exception if an UNOCCUPIED slot is not found
      array[index].state = OCCUPIED;
      array[index].object = obj;
      count++;
   }
}

public void withdraw(Comparable obj){
   if(count == 0) throw new ContainerEmptyException();
   int index = findObjectIndex(obj);
   if(index < 0)
      throw new IllegalArgumentException("Object not found");
   else {
      array[index].state = DELETED;
      // lazy deletion: DO NOT SET THE LOCATION TO null
      count--;
   }
}
```

Exercises

1. If a hash table is 25% full what is its load factor?

2. Given that,

   $$c(i) = i^2,$$

   for **`c(i)`** in quadratic probing, we discussed that this equation does not satisfy Property 2, in general. What cells are missed by this probing formula for a hash table of size 17? Characterize using a formula, if possible, the cells that are not examined by using this function for a hash table of size n.

3. It was mentioned in this session that secondary clusters are less harmful than primary clusters because the former cannot combine to form larger secondary clusters. Use an appropriate hash table of records to exemplify this situation.