Hashing

Lecture 16-18

Preview

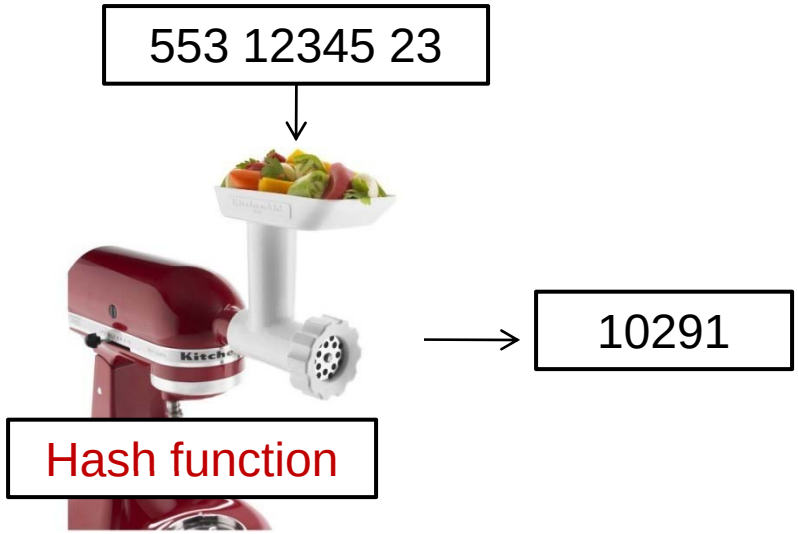
- A hash function is a function that:
  - When applied to an Object, returns a number
  - When applied to *equal* Objects, returns the *same* number for each
  - When applied to *unequal* Objects, is *very unlikely* to return the same number for each
- Hash functions turn out to be very important for searching, that is, looking things up fast

Searching

- Consider the problem of searching an array for a given value
  - If the array is not sorted, the search requires O(n) time
    - If the value isn't there, we need to search all n elements
    - If the value is there, we search n/2 elements on average
  - If the array is sorted, we can do a binary search
    - A binary search requires O(log n) time
    - About equally fast whether the element is found or not
  - It doesn't seem like we could do much better
    - How about an O(1), that is, constant time search?
    - We can do it *if* the array is organized in a particular way

Hashing

- Suppose we were to come up with a “magic function” that, given a value to search for, would tell us exactly where in the array to look
- This function would have no other purpose
- This function is called a hash function because it “makes hash” of its inputs



(Matuszek)

Hash Function

- A hash function *h* can transform a particular key *K*, be it a string, number, record, or the like, into an index in the table used for storing items of the same type as *K*
- If *h* transforms different keys into different numbers, it is called a *perfect hash function*

(Drozdek, 2005)

Example (ideal) hash function

- Suppose our hash function gave us the following values:

```
h("apple") = 5
h("watermelon") = 3
h("grapes") = 8
h("cantaloupe") = 7
h("kiwi") = 0
h("strawberry") = 9
h("mango") = 6
h("banana") = 2
```

0	kiwi
1	
2	banana
3	watermelon
4	
5	apple
6	mango
7	cantaloupe
8	grapes
9	strawberry

Sets and tables

- Sometimes we just want a set of things—objects are either in it, or they are not in it
- Sometimes we want a map—a way of looking up one thing based on the value of another
  - We use a *key* to find a place in the map
  - The associated *value* is the information we are trying to look up
- Hashing works the same for both sets and maps

. . .	*key*	*value*
141		
142	robin	robin info
143	sparrow	sparrow info
144	hawk	hawk info
145	seagull	seagull info
146		
147	bluejay	bluejay info
148	owl	owl info

Hash Function

- A good hash function should make any given key equally likely to hash to any of the m slots of the hash table (this is called simple uniform hashing)
- We will discuss some types of hash functions
  - Modulus hashing
  - Folding
  - Multiplicative hashing

(Prasitjutrakul, 2550)

Modulus

- *h(K) = K mod m where m is size of table*
- *m should not*
  - *be 10^q because it will use only q rightmost digits if key is base 10*
  - be *2^q because it will use only q rightmost digits*
  - *has a common divider c with K because K % m will yield answer as a multiple of c. If c is small, many keys will have K % m as a multiple of c resulting in small spread of key*
- *m should be prime*

(Prasitjutrakul, 2550)

Folding

- key is divided into several parts. These parts are combined or folded together and are often transformed in a certain way to create the target address
- For example, a social security number (SSN) 123-45-6789 can be divided into three parts, 123, 456, 789, and then these parts can be added. The resulting number, 1,368, can be % m

(Drozdek, 2005)

Multiplicative hashing

- Use a formula

$$h(x) = \lfloor m(xA - \lfloor xA \rfloor) \rfloor$$

where x is a key, A is any real number between (0,1) and m is size of table

- It has been suggested that a good A value is

(sqrt(5) – 1)/2 ≈ 0.618

(Prasitjutrakul, 2550)

Finding the Perfect Hash Function

- How can we come up with this magic function?
- In general, we cannot--there is no such magic function ☹
  - In a few specific cases, where all the possible values are known in advance, it has been possible to compute a perfect hash function
- What is the next best thing?
  - A perfect hash function would tell us exactly where to look
  - In general, the best we can do is a function that tells us where to *start* looking!

(Matuszek)

Example imperfect hash function

- Suppose our hash function gave us the following values:
  - h("apple") = 5
    h("watermelon") = 3
    h("grapes") = 8
    h("cantaloupe") = 7
    h("kiwi") = 0
    h("strawberry") = 9
    h("mango") = 6
    h("banana") = 2
    h("honeydew") = 6

• Now what?

(Matuszek)

Collisions

- When two keys hash to the same array location, this is called a collision
- Collisions are normally treated as “first come, first served”—the first value that hashes to the location gets it
- We have to find something to do with the second and subsequent values that hash to this same location

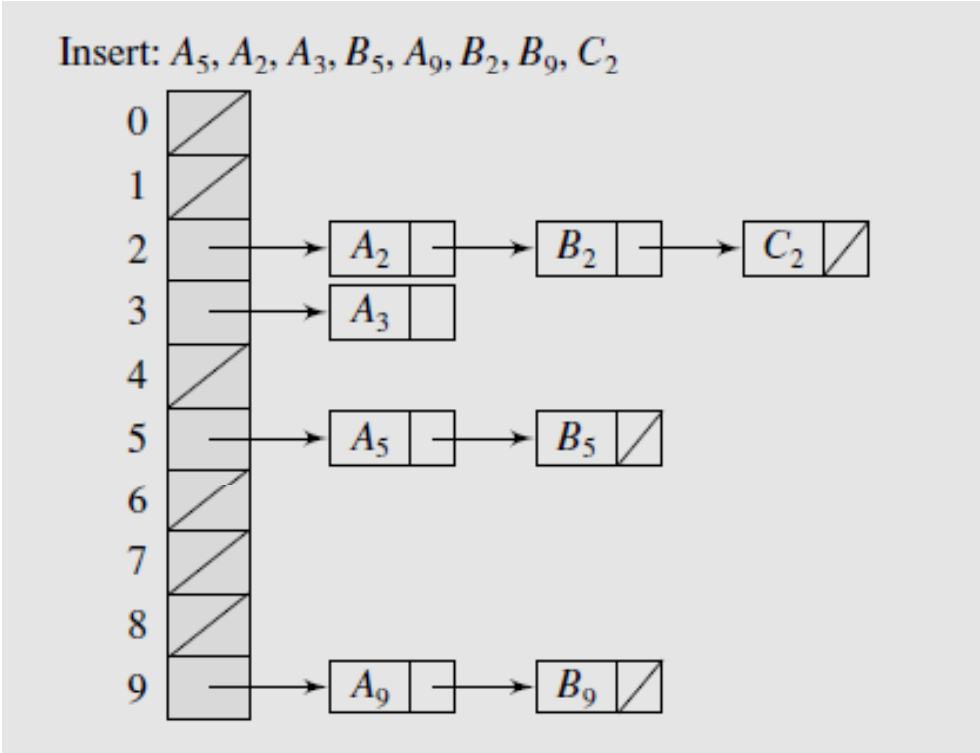
(Matuszek)

Handling collisions

- What can we do when two different values attempt to occupy the same place in an array?
  - **Solution #1:** Use the array location as the header of a linked list of values that hash to this location (Separate chaining)
  - **Solution #2:** Search from there for an empty location (Open addressing)
    - Can stop searching when we find the value *or* an empty location
    - We will learn 3 methods to find an available table entry
      - Linear probing
      - Quadratic probing
      - Double hashing
- Whichever solutions we used, remember that
  - We use the same technique to *add* things to the array as we use to *search* for things in the array

Separate Chaining

- In *chaining,* each position of the table is associated with a linked list and colliding keys are put in the same linked list



Subscripts indicate the home positions of the keys being hashed

(Drozdek, 2005)

Map Interface

- size: returns the number of key-value mappings in this map
- isEmpty: returns true if this map contains no key-value mappings
- containsKey: returns true if this map contains a mapping for the specified key
- get: returns the value to which the specified key is mapped, or null if this map contains no mapping for the key
- put: add (key, value) in map
- remove: remove mappings that contain the given key

- Interface

```
public interface Map {
    public int size();
    public boolean isEmpty();
    public boolean containsKey(Object key);
    public Object get(Object key);
    public Object put(Object key, Object
value);
    public void remove(Object key);
}
```

Map contains (key, value) where key is unique

(Prasitjutrakul, 2550)

Class: SeparateChainingHashMap (1)

```
public class SeparateChainingHashMap implements Map{
    private static class LinkedNode {
        Object key, value;
        LinkedNode next;
        LinkedNode(Object k, Object v, LinkedNode n) {
            key = k; value = v; next = n;
        }
    }

    private int size;
    private LinkedNode[] table;
    public SeparateChainingHashMap(int cap) {
        table = new LinkedNode[cap];
    }

    public int size() { return size; }
    public boolean isEmpty() { return size==0; }
    //continue next slide…
```


(Prasitjutrakul, 2550)

Class: SeparateChainingHashMap (2)

```
public Object get(Object key) {
    LinkedNode node = getNode(key);
    return node ==null?null:node.value;
}
public boolean containsKey(Object key) {
    return getNode(key)!=null;
}
private LinkedNode getNode(Object key) {
    LinkedNode curr = table[h(key)];
    while(curr!=null && !curr.key.equals(key)) {
        curr = curr.next;
    }
    return curr;
}
private int h(Object x) {
    return Math.abs(x.hashCode())%table.length;
}
//continue next slide
```


(Prasitjutrakul, 2550)

Class: SeparateChainingHashMap (3)

```
public Object put(Object key, Object value) {
    LinkedNode node = getNode(key);
    Object oldValue = null;
    if(node!=null) {
        oldValue = node.value;
        node.value = value;
    } else {
        int h = h(key);
        table[h] = new LinkedNode(key,value,table[h]);
        ++size;
    }
    return oldValue;
}
//continue next slide
```


(Prasitjutrakul, 2550)

Class: SeparateChainingHashMap (4)

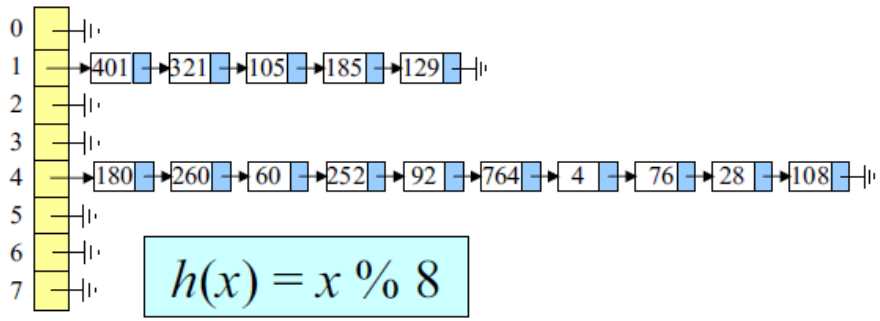
```
    public void remove(Object key) {
        int h = h(key);
        if(table[h]==null) return;
        if(table[h].key.equals(key)) {
            table[h] = table[h].next; --size;
        } else {
            LinkedNode prev = table[h];
            while(prev.next!=null && !prev.next.key.equals(key)) {
                prev = prev.next;
            }
            if(prev.next !=null) {
                prev.next = prev.next.next;
                --size;
            }
        }
    }
}
```


(Prasitjutrakul, 2550)

Spread of Data

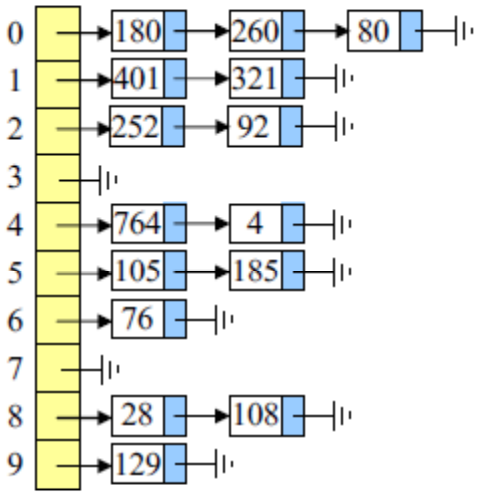
- Key and hash function determine the spread of data
- If the data are spread nicely/evenly, each cell should contain a list of length λ
  - Where λ is number of elements in the table / table size
  - λ is called load factor
  - If λ is small, a look up is fast
  - Searching and deletion take O(λ)
- If the data are spread poorly,

  some cell contain a list much longer than λ
  - Searching is slow as searching in list



$h(x) = x \% 8$

$h(x) = x \% 10$



(Prasitjutrakul, 2550)

Open Addressing

- In the open addressing method, when a key collides with another key, the collision is resolved by finding an available table entry other than the position (address) to which the colliding key is originally hashed
- If position *h(K)* is occupied, then the positions in the probing sequence (order of places looked at in the table) are tried until either an available cell is found or the same positions are tried repeatedly or the table is full

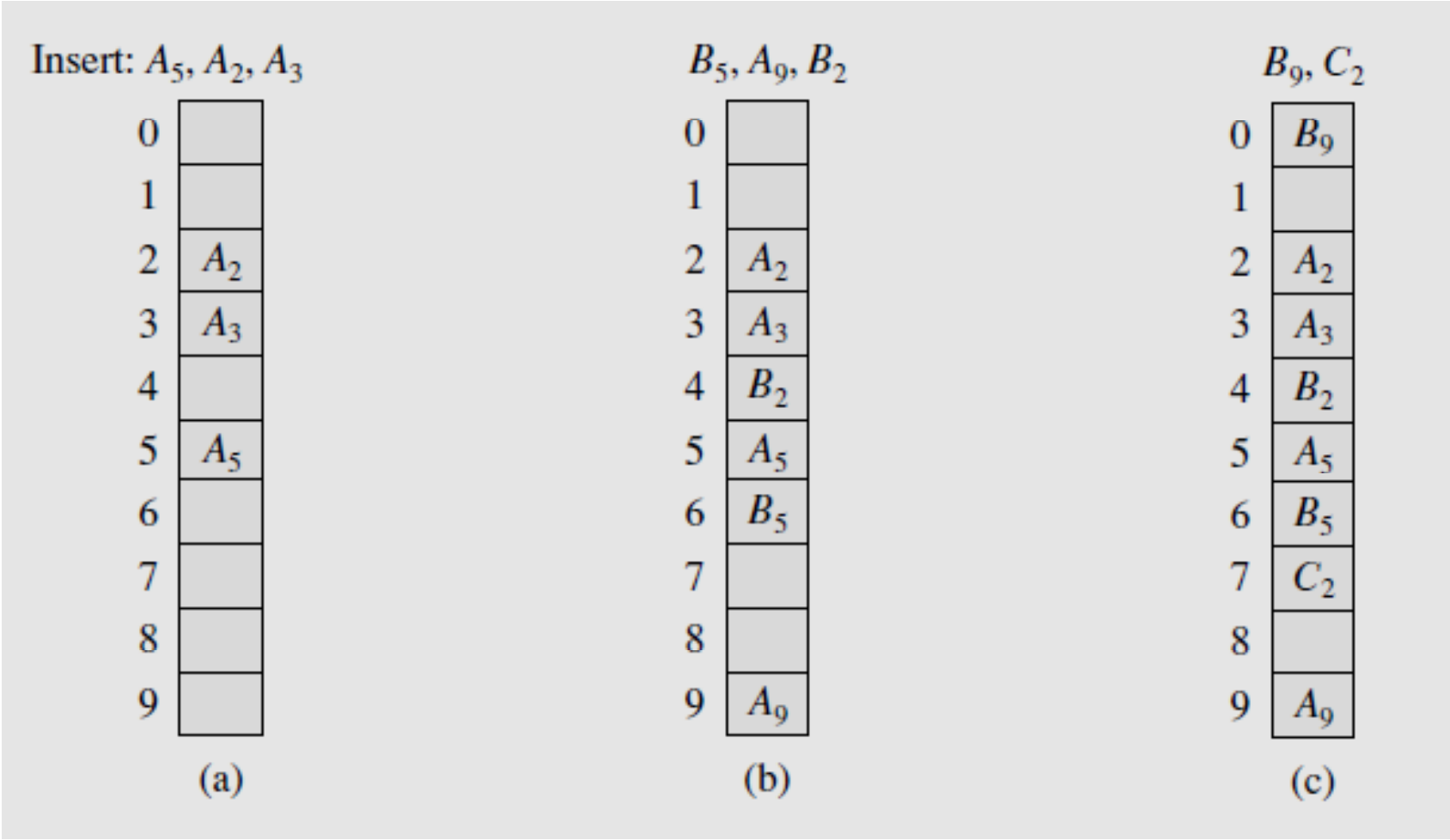
(Drozdek, 2005)

Linear Probing (1)

- In linear probing, sequentially searching all positions starting from the position calculated by the hash function until an empty cell is found.
- If the end of the table is reached and no empty cell has been found, the search is continued from the beginning of the table
- In the extreme case—empty cell is the cell preceding the one from which the search started
- The position to be tried is $h_j(K) = (h(K) + j) \% m$, where $h_j(K)$ is a place to look at (index) after j^{th} colision, $h(K)$ is home address and m is table size
  - *Which can be rewrite as* $h_j(K) = (h_{j-1}(K) + 1) \% m$

(Drozdek, 2005)

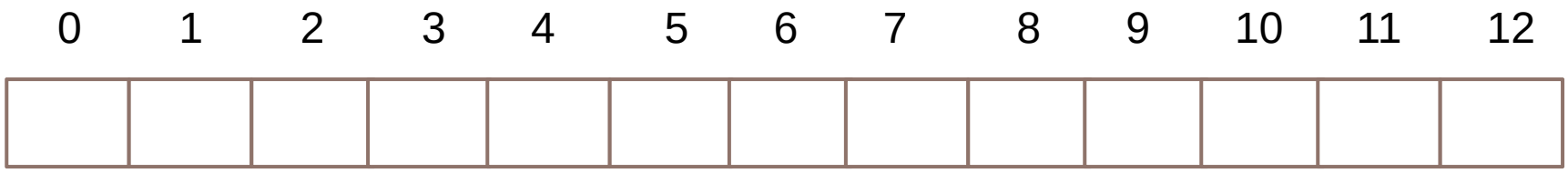
Linear Probing (2)



(Drozdek, 2005)

Linear Probing (3)

- h(K) = K % 13
- Keys: 22, 37, 12, 26, 35, 9



22 => h(K) => 9
37 => h(K) =>11
12 => h(K) => 12
26 => h(K) => 0
35 => h(K) => 9
9 => h(K) => 9

Class: LinearProbingHashMap (1)

```
public class LinearProbingHashMap implements Map {

    private static class Entry {
        Object key, value;
        Entry(Object k, Object v) {
            key = k;
            value = v;
        }
    }
    private Entry[] table;
    private int size;

    public LinearProbingHashMap(int m) {
        table = new Entry[m];
    }

    public int size() { return size; }
    public boolean isEmpty() { return size == 0; }
    //continue next slide
```

Class: LinearProbingHashMap (2)

```
public boolean containsKey(Object key) {
      return table[indexOf(key)] != null;
  }

  private int indexOf(Object key) {
       int h = h(key);
      for (int j = 1; j < table.length; j++) {
          if (table[h] == null) {
              return h;
          }
          if (table[h].key.equals(key)) {
              return h;
          }
          h = (h + 1) % table.length;
      }
      throw new AssertionError("A table is full!");
  }
  //continue next slide
```


(Prasitjutrakul, 2550)

Class: LinearProbingHashMap (3)

```
public int h(Object key) {
    return Math.abs(x.hashCode()) % table.length;
}

public Object get(Object key) {
    Entry e = table[indexOf(key)];
    return e == null ? null : e.value;
}
//continue next slide
```

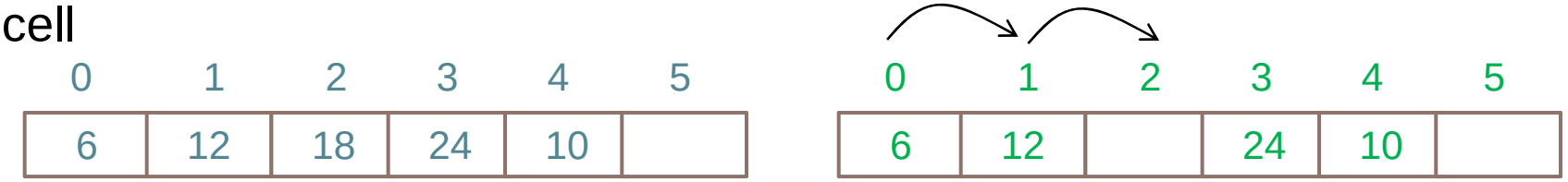

(Prasitjutrakul, 2550)

Class: LinearProbingHashMap (4)

```
public Object put(Object key, Object value) {
    Object oldValue = null;
    int i = indexOf(key);
    if (table[i] == null) {
        table[i] = new Entry(key, value);
        ++size;
    } else {
        oldValue = table[i].value;
        table[i].value = value;
    }
    return oldValue;
}
//continue next slide
```

Class: LinearProbingHashMap (5)

- We cannot simply put null in the cell we want to remove the element, because we will not be able to find key that has been probed to this cell



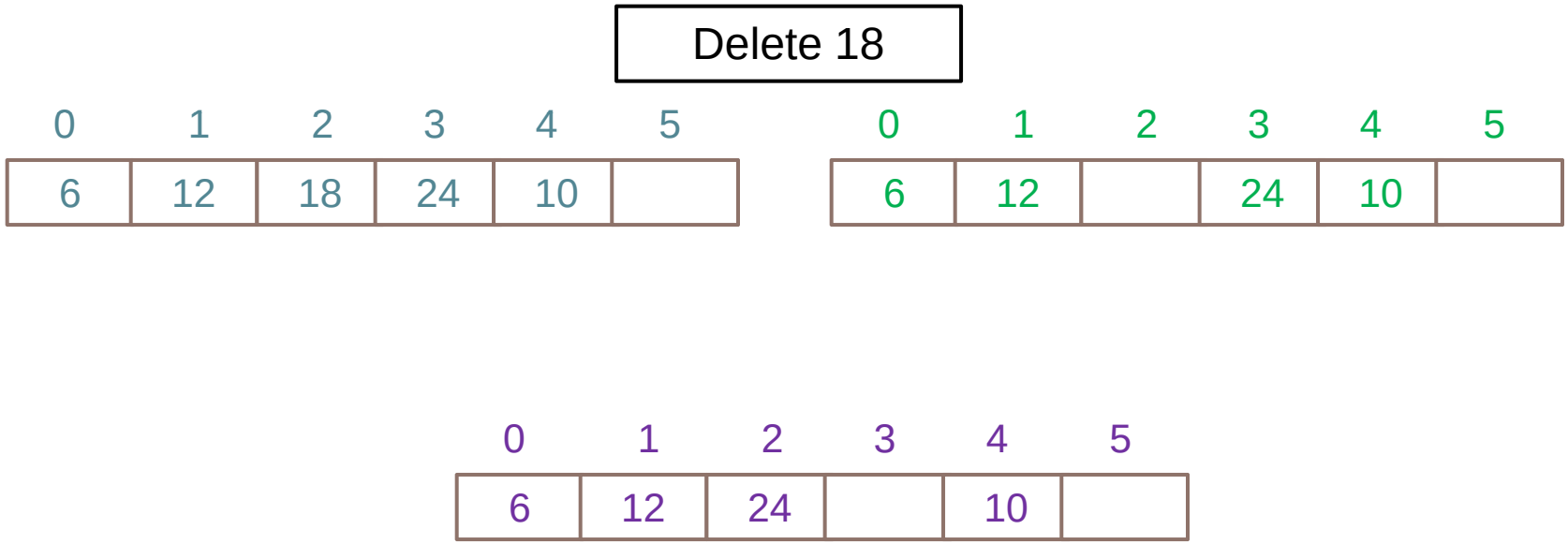

```
public void remove(Object key) {
    int i = indexOf(key);
    if (table[i] != null) {
        table[i] = null;
        --size;
        for (++i; table[i] != null; i = (i + 1) % table.length) {
            Entry e = table[i];
            table[i] = null;
            table[indexOf(e.key)] = e;
        }
    }
}
```

H(K) = K%6

(Prasitjutrakul, 2550)

Method: remove

- Need to rehash 24 and 10
- We stop at 10 because the next cell is empty



(Prasitjutrakul, 2550)

Rehash (1)

- When λ is big, the number of searches increase. Therefore, we should control λ to not exceed a certain value in order to control the time needed
- We exchange memory for speed

```
public Object put(Object key, Object value) {
        Object oldValue = null;
        int i = indexOf(key);
        if (table[i] == null) {
            table[i] = new Entry(key, value);
            ++size;
            if(size > table.length/2) rehash();
        } else {
            oldValue = table[i].value;
            table[i].value = value;
        }
        return oldValue;
    }
```


(Prasitjutrakul, 2550)

Rehash (2)

```
private void rehash() {
    Entry[] oldT = table;
    table = new Entry[2*table.length];
    for(int i=0; i < oldT.length; i++) {
        if(oldT[i] != null) table[indexOf(oldT[i].key)] = oldT[i];
    }
}
```


(Prasitjutrakul, 2550)

Primary Cluster

- One problem with the above technique is the tendency to form “clusters”
- A cluster is a group of items not containing any open slots
- The bigger a cluster gets, the more likely it is that new values will hash into the cluster, and make it ever bigger
- Clusters cause efficiency to degrade
- Here is a *non*-solution: instead of stepping one ahead, step n locations ahead (quadratic probing)
  - The clusters are still there, they’re just harder to see

Quadratic Probing (1)

- *In quadratic probing, the position tried is*

  $h_j(K) = (h(K) + (-1)^{j-1}((j + 1)/2)^2) \% m$

  for $j = 1, 2, \ldots, m - 1$

- *This cumbersome formular can be expressed as a sequence of probes:*

  $h(K) + j^2, h(K) - j^2$ *for* $j = 1, 2, \ldots, (m - 1)/2$

- *Including the first attempt to hash K, this results in sequence:*

  $h(K), h(K) + 1, h(K) - 1, h(K) + 4, h(K) - 4, \ldots, h(K) + (m - 1)^2/4,$ h(K) $- (m - 1)^2/4$

(Drozdek, 2005)

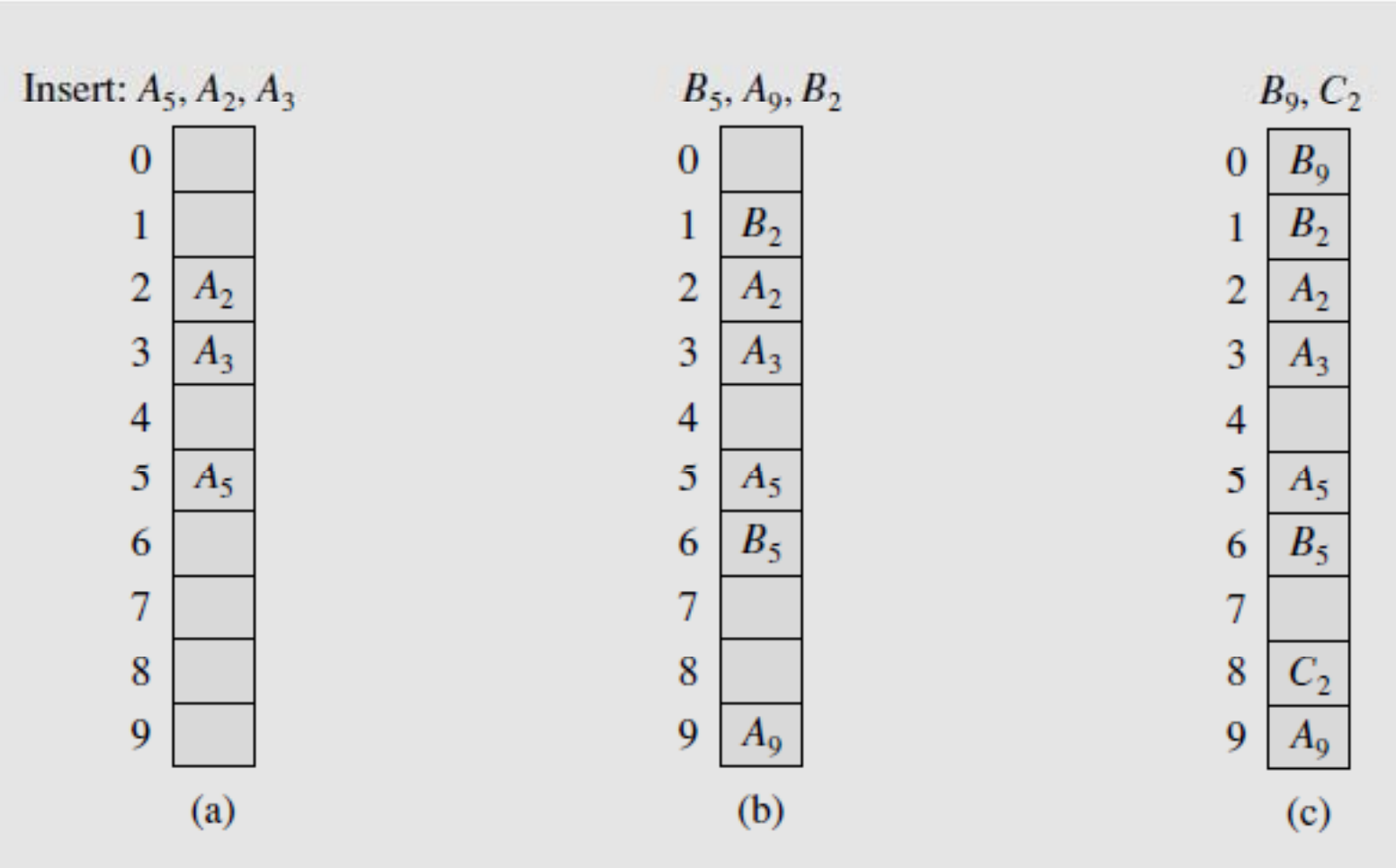
Quadratic Probing (2)

- Ideally, the table size should be a prime $4j + 3$ of an integer j, which guarantees the inclusion of all positions in the probing sequence (Radke, 1970)
- If $m = 19$, and assuming that $h(K) = 9$ for some K, the resulting sequence of probes is

  9, 10, 8, 13, 5, 18, 0, 6, 12, 15, 3, 7, 11, 1, 17, 16, 2,14, 4

(Drozdek, 2005)

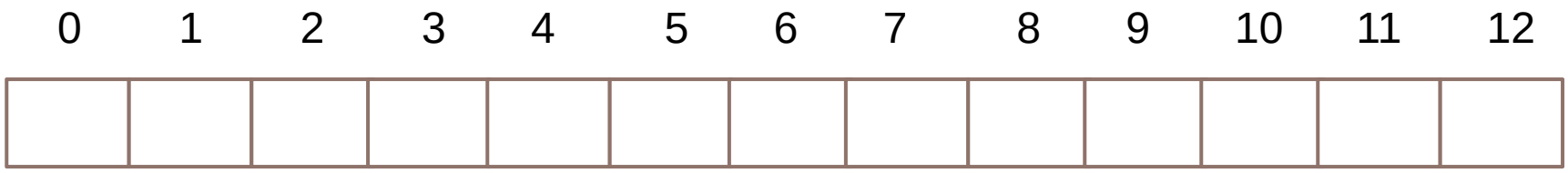
Quadratic Probing (3)



(Drozdek, 2005)

Quadratic Probing (3)

- h(K) = K % 13
- Keys: 22, 37, 12, 26, 35, 9



22 => h(K) => 9
37 => h(K) =>11
12 => h(K) => 12
26 => h(K) => 0
35 => h(K) => 9
9 => h(K) => 9

(Prasitjutrakul, 2550)

Class: QuadraticProbingHashMap (1)

```
import java.math.BigInteger;

public class QuadraticProbingHashMap implements Map{

    private static class Entry {…}
    private static final Entry DELETED = new Entry(new Object(), null);
    private Entry[] table;
    private int size;
    private int numNonNulls;

    public QuadraticProbingHashMap(int m) {
        table = new Entry[m];
    }
    public int size() {…}
    public boolean isEmpty() {…}
    public boolean containsKey(Object key) {…}
    public Object get(Object key) {…}
     public int h(Object key) {…}
    //continue next slide
```

{…} means same as in LinearProbingHashMap

(Prasitjutrakul, 2550)

Class: QuadraticProbingHashMap (2)

```
private int indexOf(Object key) {
    int h = h(key);
    int hi = h;
    for(int j=1; j < table.length; j++) {
        if(table[hi]==null) break;
        if(table[hi].key.equals(key)) break;
        hi = (int) (h + Math.pow(-1,j-1)*Math.pow((j+1)/2,2)) % table.length;
        if(hi < 0) hi = hi+table.length;
    }
    return hi;
}
//continue next next slide
```

Lazy deletion

- Can no longer use the same remove logic as in linear probing because probing sequence is not contiguous anymore
- Use another approach called "lazy deletion"
  - We simply put an object, which we will call DELETED, to the cell we want to remove
  - DELETED has a unique key that is guaranteed that it is not equal any other keys
  - Once load factor is more than 0.5, need to rehash to remove DELETED
- Each cell could be
  - Empty cell; table[i] == null
  - Contained an element but was removed; table[i] == DELETED
  - Contain an element; table[i] != null && table[i] != DELETED
- This approach can be used in linear probing too

Class: QuadraticProbingHashMap (3)

```
public void remove(Object key) {
    int i = indexOf(key);
    if(table[i] != null) {
        table[i] = DELETED;
        --size;
    }
}
//continue next slide
```

Class: QuadraticProbingHashMap (4)

```
public Object put(Object key, Object value) {
    Object oldValue = null;
    int i = indexOf(key);
    if(table[i] == null) {
        table[i] = new Entry(key, value);
        ++size; ++numNonNulls;
        if(numNonNulls > table.length/2) rehash();
    } else {
        oldValue = table[i].value;
        table[i].value = value;
    }
    return oldValue;
}
//continue next slide
```

Class: QuadraticProbingHashMap (5)

```
    private void rehash() {
        Entry[] oldT = table;
        table = new Entry[nextPrime(4*size)];
        for(int i =0; i < oldT.length; i++) {
            if(oldT[i] != null && oldT[i] !=DELETED) {
                int j = indexOf(oldT[i].key);
                table[j] = oldT[i];
            }
        }
        numNonNulls = size;
    }

    private int nextPrime(int m) {
        BigInteger b = new BigInteger(Integer.toString(m));
        return b.nextProbablePrime().intValue();
    }
}
```


(Prasitjutrakul, 2550)

Secondary Cluster

- In quadratic probing, the problem of cluster buildup is not avoided altogether, because for keys hashed to the same location, the same probe sequence is used. Such clusters are called *secondary clusters*
- The problem of secondary clustering is best addressed with *double hashing*

(Prasitjutrakul, 2550)

Double Hashing (1)

- Double hashing utilizes two hash functions, one for accessing the primary position of a key, *h,* and a second function, h_p*, for resolving conflicts.*
- The position tried is $h_j(K) = (h(K) + j * h_p(K)) \% m$ where $h_j(K)$ is place to look at after j^{th} *colision,* $h(K)$ is home address, $h_p(K)$ is another hash function used to determine distance between probes and m is table size
  - Which can be rewrite as $h_j(K) = (h_{j-1}(K) + h_p(K)) \% m$
- if *K1* and *K2* are hashed primarily to the same position *j,* the *probing sequences* can be different for each. This, however, depends on the choice of the second hash function, h_p*,* which may render the same sequences for both keys.
  - This is the case for function $h_p(K) = K.length$ when both keys are of the same length

(Drozdek, 2005)

Double Hashing (2)

- $h_p(K)$ % m != 0 otherwise probing is at the same cell
- $h_p(K)$ and m should not have a common divider (except 1) otherwise probing sequence repeats itself
  - E.g. m=12, $h_p(K)$ = 9, h(K) = 3
  - Probing sequence is 3, (3+9)%12 -> 0, (3+18)%12 -> 9, (3+27)%12 -> 6, (3+36)%12 -> 3, (3+45)%12 -> 0, (3+54)%12 -> 9, (3+63)%12 -> 6, (3+72)%12 -> 3
- m should be prime

(Prasitjutrakul, 2550)

Class: DoubleHashingHashMap

- All the details in DoubleHashing class is the same as in QuadraticProbingHashMap **except** indexOf and g method

```
private int indexOf(Object key) {
        int h = h(key);
        int g = g(key);
        for(int j=1; j < table.length; j++) {
            if(table[h]==null) break;
            if(table[h].key.equals(key)) break;
            h = (h+g) % table.length;
        }
        return h;
    }
    private int g(Object key){
        return 1 + Math.abs(x.hashCode()) % (table.length/2);
    }
```


(Prasitjutrakul, 2550)

Clustering Comparison

Linear Probing

Quadratic Probing

Double Hashing

(Prasitjutrakul, 2550)

	Linear Probing		Quadratic Probing		Double Hashing	
LF	Successful	Unsuccessful	Successful	Unsuccessful	Successful	Unsuccessful
0.05	1.0	1.1	1.0	1.1	1.0	1.1
0.10	1.1	1.1	1.1	1.1	1.1	1.1
0.15	1.1	1.2	1.1	1.2	1.1	1.2
0.20	1.1	1.3	1.1	1.3	1.1	1.2
0.25	1.2	1.4	1.2	1.4	1.2	1.3
0.30	1.2	1.5	1.2	1.5	1.2	1.4
0.35	1.3	1.7	1.3	1.6	1.2	1.5
0.40	1.3	1.9	1.3	1.8	1.3	1.7
0.45	1.4	2.2	1.4	2.0	1.3	1.8
0.50	1.5	2.5	1.4	2.2	1.4	2.0
0.55	1.6	3.0	1.5	2.5	1.5	2.2
0.60	1.8	3.6	1.6	2.8	1.5	2.5
0.65	1.9	4.6	1.7	3.3	1.6	2.9
0.70	2.2	6.1	1.9	3.8	1.7	3.3
0.75	2.5	8.5	2.0	4.6	1.8	4.0
0.80	3.0	13.0	2.2	5.8	2.0	5.0
0.85	3.8	22.7	2.5	7.7	2.2	6.7
0.90	5.5	50.5	2.9	11.4	2.6	10.0
0.95	10.5	200.5	3.5	22.0	3.2	20.0

(Drozdek, 2005)

Number of Trials

- Formulas approximating, for different hashing methods, the average numbers of trials for successful and unsuccessful searches

	Linear Probing	Quadratic Probing[a]	Double Hashing
successful search	$\frac{1}{2}\left(1+\frac{1}{1-LF}\right)$	$1-\ln(1-LF)-\frac{LF}{2}$	$\frac{1}{LF}\ln\frac{1}{1-LF}$
unsuccessful search	$\frac{1}{2}\left(1+\frac{1}{(1-LF)^2}\right)$	$\frac{1}{1-LF}-LF-\ln(1-LF)$	$\frac{1}{1-LF}$

Load Factor $LF = \frac{\text{number of elements in the table}}{\text{table size}}$

[a] The formulas given in this column approximate any open addressing method that causes secondary clusters to arise, and quadratic probing is only one of them.

(Knuth, 1998)

Summary

- Efficiency of hash table depends on load factor
- In separate chaining, number of element (n) can be > table size (m), while in open addressing n <= m
- In separate chaining, groups having high collision will not effect groups having low collision
- In open addressing, groups having high collision will effect other group
- Hash table does not work well with problems that require data ordering e.g. sorting, finding the largest number

References

- Drozdek, Adam. “Hashing." *Data Structures and Algorithms in Java. Boston, MA: Thomson Course Technology, 2005. 519*-569. Print
- Matuszek, D. Hashing [PDF document]. Retrieved from Lecture Notes Online Web site: http://www.cis.upenn.edu/~matuszek/cit594-2013/index.html
- Prasitjutrakul, S. “ตารางแฮช.” *โครงสร้างข้อมูล ฉบับวาจาจาวา. กรุงเทพฯ: สำนักพิมพ์แห่งจุฬาลงกรณ์ มหาวิทยาลัย, 2550. 247-279. Print.*