Reminders and Topics

- **Search project due Friday**
  - Last project, Yay!
- **SRTI Course Survey - online**
  - Survey period: 12/3 7am - 12/20 11pm
- This lecture:
  - **Hashing and Hash Tables**
- Next lecture:
  - **Final review**

So Far We've Learned

data structure	add	search	remove
unsorted array	`O(1)`	`O(N)`	`O(N)`
unsorted linked	`O(1)`	`O(N)`	`O(N)`
sorted array	`O(N)`	`O(log N)`	`O(N)`
BST (balanced / worst)	`O(log N) / O(N)`	`O(log N) / O(N)`	`O(log N) / O(N)`
heap	`O(log N)`	`O(N)`	`O(log N)`

Is this the best? Can we do better??

Hash Table

- **Surprisingly Fast**
  - On average, O(1) for all of add/search/delete!
- **Surprisingly Simple**
  - Easy to implement
- **No way, it can’t perfect. What’s the catch?**
  - Need to have a good idea about the number of elements. Difficult to re-size dynamically.
  - Performance degrades when it’s close to full.
  - Can’t easily visit data in sorted order (such as getting the max).

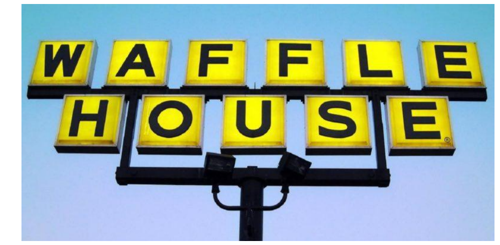
A First Idea

- Here is a trivial way to achieve O(1) cost: say we want to store and search student records based on student ID, and we assume each ID is unique.
- Since each ID is 8-digit long, let's prepare an array / table with a capacity of 100,000,000, so there is one entry reserved for each possible ID (0 to 99,999,999).
- Add / Search / Remove all cost just O(1)!
- What's the problem with this approach?

Not all 8-digit integers are valid student IDs!

Many entries are empty. This is a huge waste of storage!

Hashing

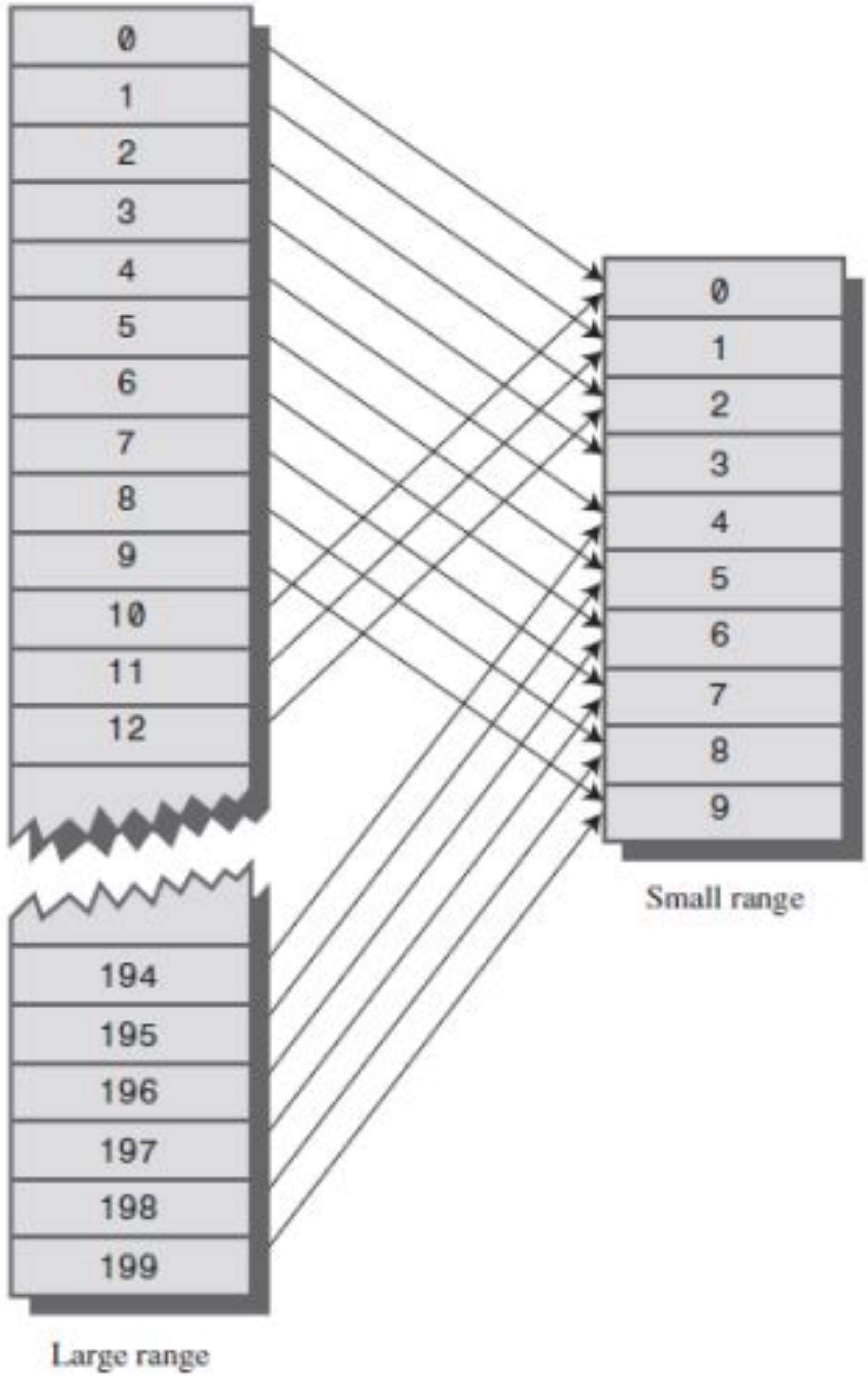


- UMass has about 22,000 undergrads (way less than the number of 8-digit integers). Ideally we want a table that's not too much larger than that.
- How do we know where each ID should be stored at?
- **Hashing** comes to rescue:
  - What does the word **hash** mean? Hash browns?
  - Hashing maps a key value (which can span a wide range) to an index (which has a much smaller range).
  - Sparsity is characteristic of such data
    - credit card numbers, dictionary words

Hashing

- How do we map the huge range of possible key values to a much smaller range, so that they can be stored in a reasonably sized array?
  - Example: map an 8-digit key to 50,000 size array. There can be many different ways, what's the simplest?

- Use **modulo** (%): **`index = key % array_size`**
- This is called a **hash function**.
- array_size must be at least the number of elements (e.g. the number of students), but often a few times bigger


0
1
2
3
4
5
6
7
8
9
10
11
12
194
195
196
197
198
199
Large range
0
1
2
3
4
5
6
7
8
9
Small range

Collisions

- One obvious problem is that multiple keys can map to the same index (below array_size is 50000):
  - `23245467 % 50000 = 45467`
  - `43345467 % 50000 = 45467`
  - In fact, (45467 + 50000 x k) % 50000 = 45467 for any positive integer k!
  - This is called **<u>collision</u>**. It can be reduced by using a better hash function (e.g. array size should always be a prime number). But it cannot be avoided.

Collision Handling

- There are empty slots in the hash table to store collided elements, because we know the array size is at least the number of elements.
- We need a systematic way to search for such empty slots when collision happens. There are two generally approaches:
  - **Open addressing**
  - **Separate chaining**

Open Addressing

- **Linear Probing**

When **inserting** a new element, if a collision is encountered, check the succeeding slots one by one, until an empty slot is found.

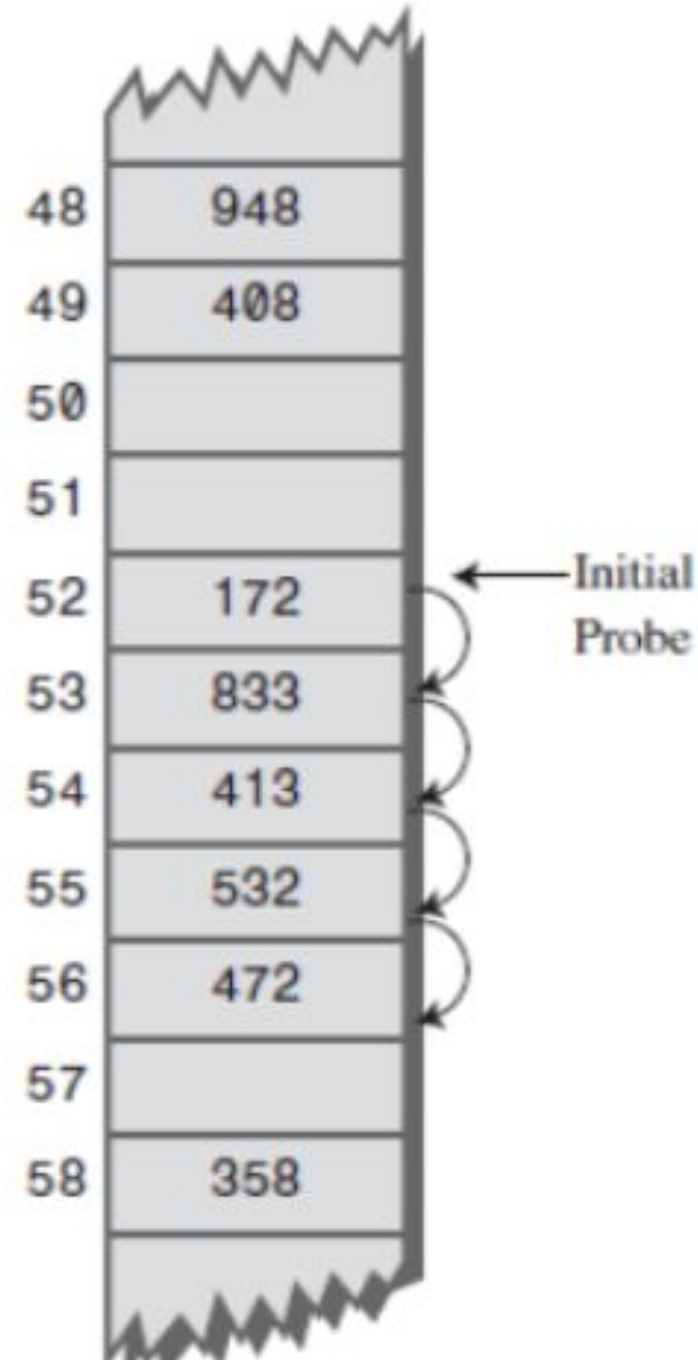
`index, index+1, index+2,` …. (the array is circular, so back to 0 when reaching the end of the array)

- Guaranteed to find one empty slot eventually.
- Demo.
- Probe length (i.e. step size) doesn't necessarily have to be 1.

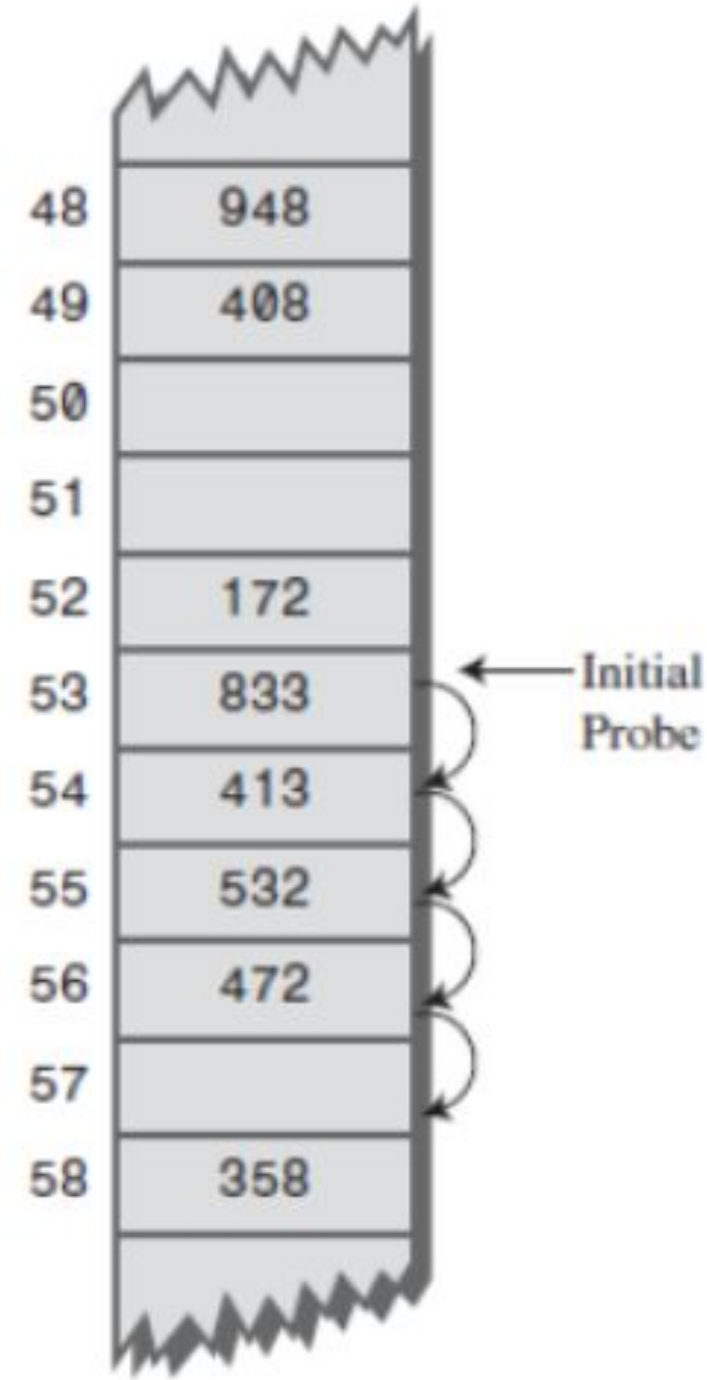
Linear Probing

For now, assume we don't remove any elements yet.

When **searching** for an element, probe linearly until either the element is found, **or an empty slot is encountered** (i.e. element does not exist). Why?



a) Successful search for 472



b) Unsuccessful search for 893

Linear Probing

- What about **deletion**?
  - Unlike before, you can’t delete an element by simply setting it to null. This will affect how you search. Why?
- How do we solve this problem?
  - Solution: **lazy deletion**. We don’t set the deleted element to null, instead, we set a special flag to indicate it’s gone. Insertion can overwrite a flagged slot; but search must continue across it.

Lazy Delete Example (index = key %10)

0	1	2	3
null	null	null	null

Insert 11, 21, then 32

0	1	2	3
null	11	21	32

Delete 21 (it gets flagged)
Search can still find 32

0	1	2	3
null	11	21	32

Insert 31

0	1	2	3
null	11	31	32

Generic Hash Table

```
class DataEntry<T>{
  public int key; // As you will see later key can also be generic...
  public T value;
  public boolean flag;
  private DataEntry(int insertKey, T insertValue){
    key = insertKey;
    value = insertValue;
    flag = false;
  }
}
class HashTable<T> {
  protected DataEntry<T>[] table;
  protected int numElements=0;
  final int DEFCAPACITY = 100;
  public HashTable(int capacity) {
    table = (DataEntry<T>[]) new Object[DEFCAPACITY];
  }
```

Linear Probing

```
// assume elements are stored in table
// the capacity of the table is table.length;
public T search (int key){
  int index = key % table.length; // hash func
  int start = index;  // keep a copy of index
  while(table[index] != null) {
    if(table[index].key == key && !table[index].flag)
        return table[index].value;
    index = (index+1) % table.length; // what's this?
    if(index==start) return null; // if back to start
  }
  return null;
}
```

Wrap around to the beginning if you've reached the end of the table. This is just like the circular array queue.

Linear Probing

```
public T delete (int key){
  int index = key % table.length; // hash func
  while(table[index] != null)
    if(table[index].key == key && !table[index].flag) {
      T value = table[index].value;
      table[index].flag = true;
      return value;
    }
    index = (index+1) % table.length;
  }
  return null;
}
```

Caveat: What happens when the item doesn’t exist?

Linear Probing

```
public void insert(int key, T value){
  int index = key % table.length; // hash func
  while(table[index] != null &&
      !table[index].flag)
    index = (index+1) % table.length;

  // new item always has a cleared flag
  table[index] = new DataItem(key, value);
}
```

Caveat: what happens when the table is full?

Clicker #1

The following hashtable has a **capacity of 10**. The index of each slot is shown below for convenience. Using **linear probing** (probe length=1), where would a new element **59** be inserted into?

60			**53**	**13**	**25**		**27**		**48**
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

A. **[1]**

B. **[2]**

C. **[6]**

D. **[8]**

E. **[9]**

Linear Probing

```
// assume elements are stored in table
// the capacity of the table is table.length;
public T search (int key){
  int index = key % table.length; // hash func
  int start = index;  // keep a copy of index
  while(table[index] != null) {
    if(table[index].key == key && !table[index].flag)
        return table[index].value;
    index = (index+probe_length) % table.length;
    if(index==start) return null; // if back to start
  }
  return null;
}
```

Probe length doesn't have to be 1. It can be any positive integer of your choice.

Clicker #2

The following hashtable has a **capacity of 10**. The index of each slot is shown below for convenience. Using **linear probing** (probe length=4), where would a new element **59** be inserted into?

60			53	13	25		27		48
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

A. [1]

B. [2]

C. [6]

D. [8]

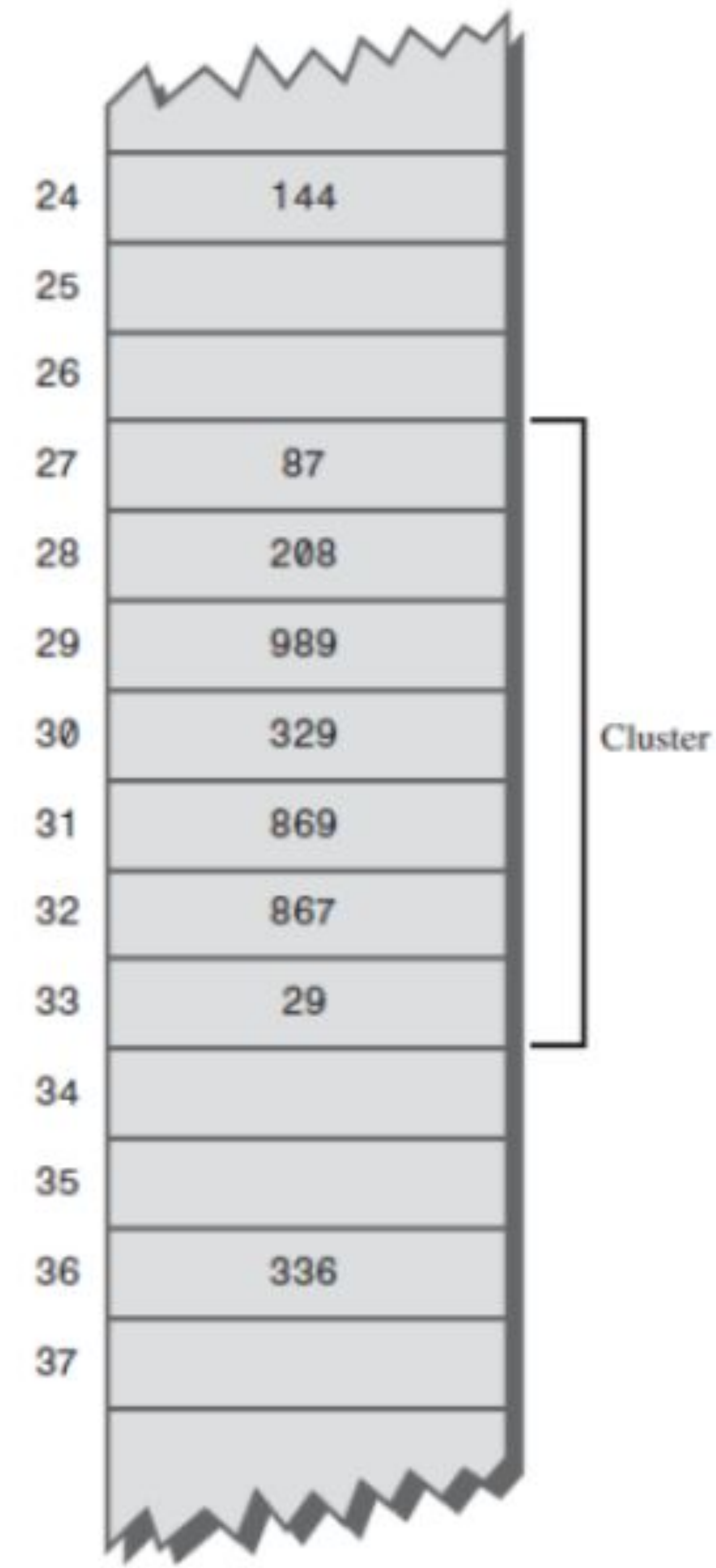
E. [9]

Linear Probing

- **Clustering problem**

Search can take a long time if a cluster is formed.

What happens when the array is close to full?



Quadratic Probing

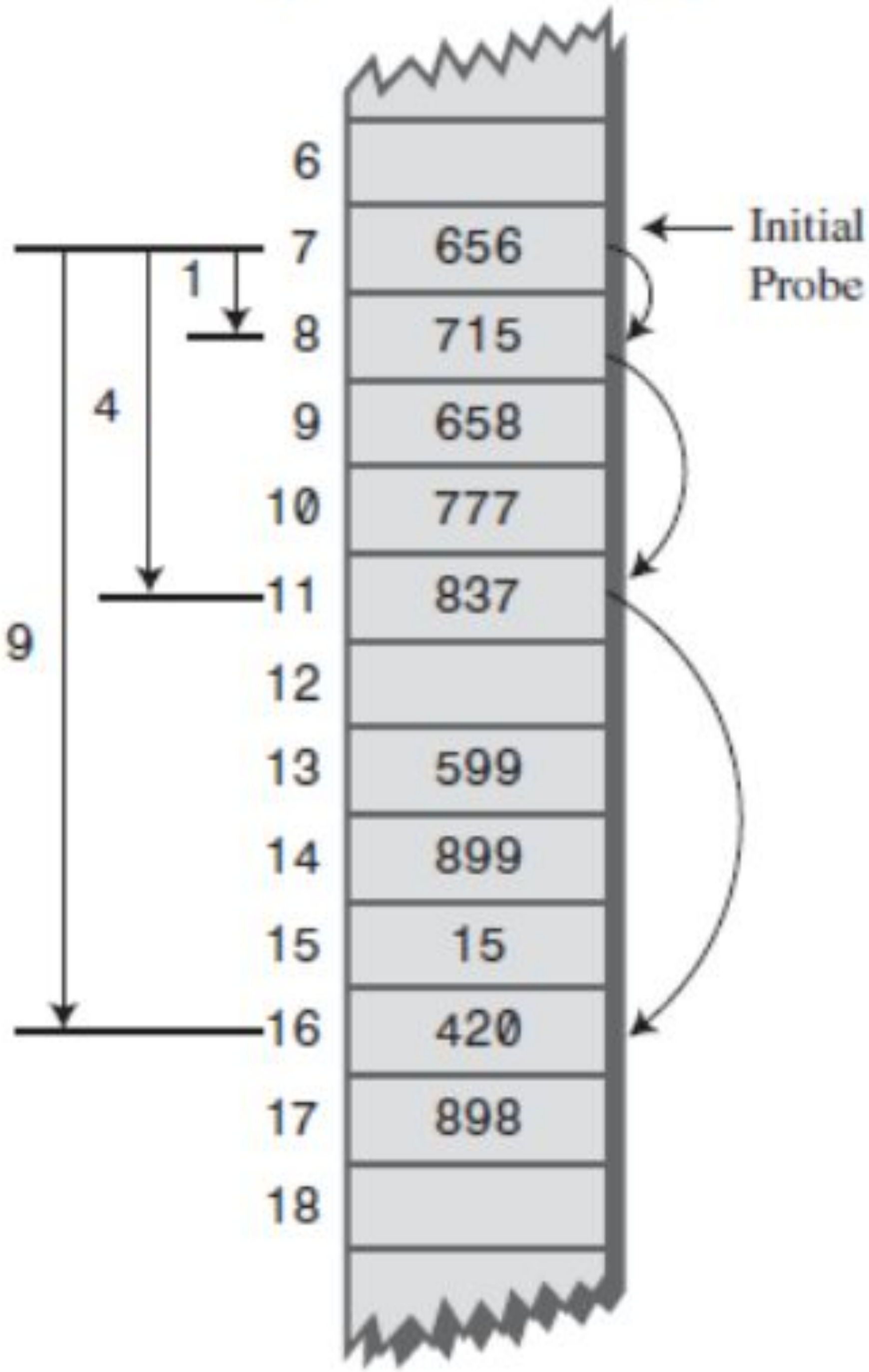
When collision happens, check the succeeding slots in **quadratic** steps / probe lengths:

`index, index+1, index+4, index+9, index+16,`

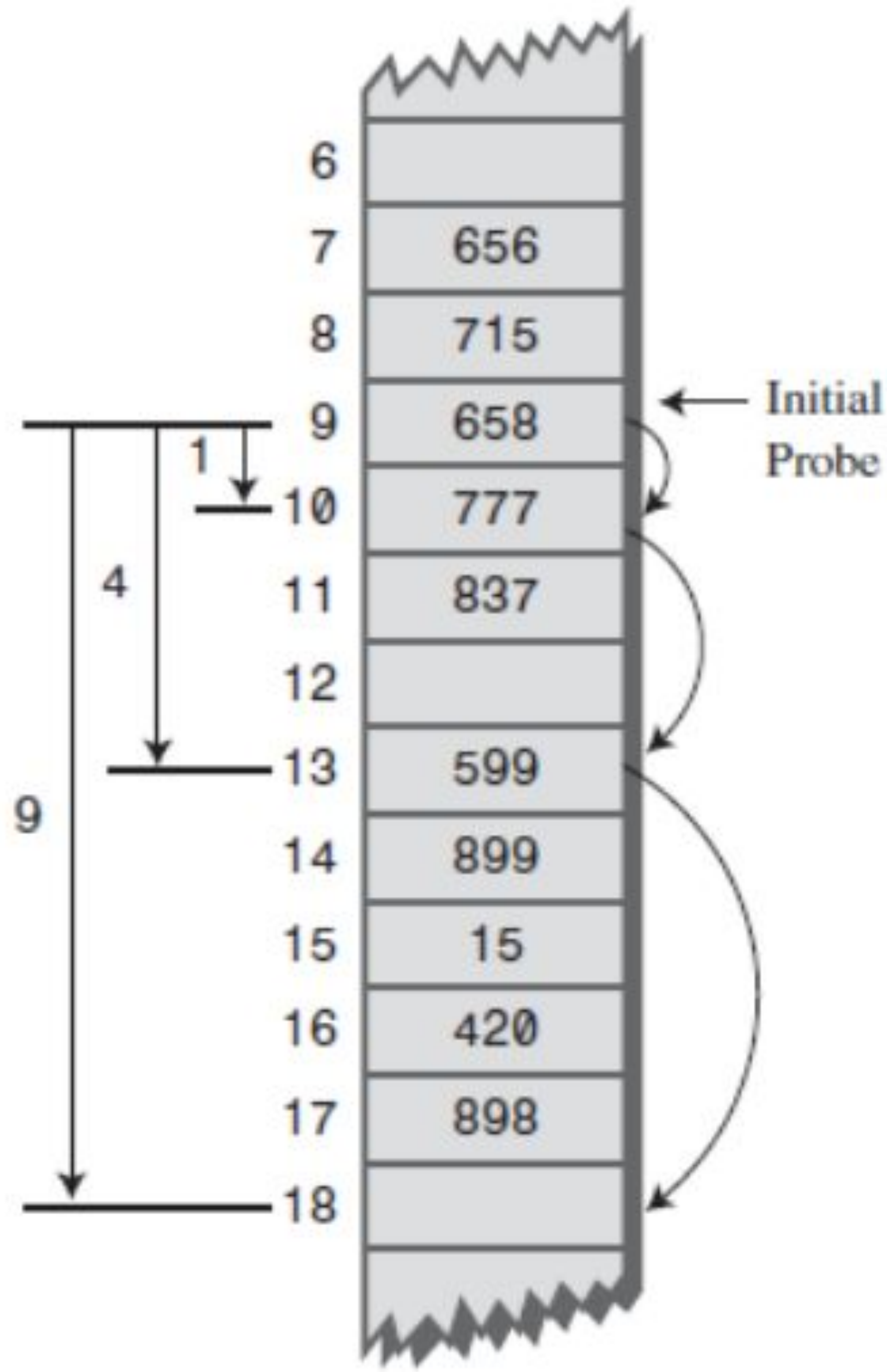
`index+25 …`

- Using increasingly larger steps helps reduce the possibility of forming clusters.

Quadratic probing



a) Successful search for 420



b) Unsuccessful search for 481

Double Hashing

- The problem with linear and quadratic probing is that once keys collide, they all follow exactly the same probing path, completely predictable.
- We need a way to generate probe lengths that can vary and are not pre-determined.
- The solution is to pick a **probe length that varies depending on the key value**. This can be achieved using a second hashing function, thus the name ‘double hashing’.

Double Hashing

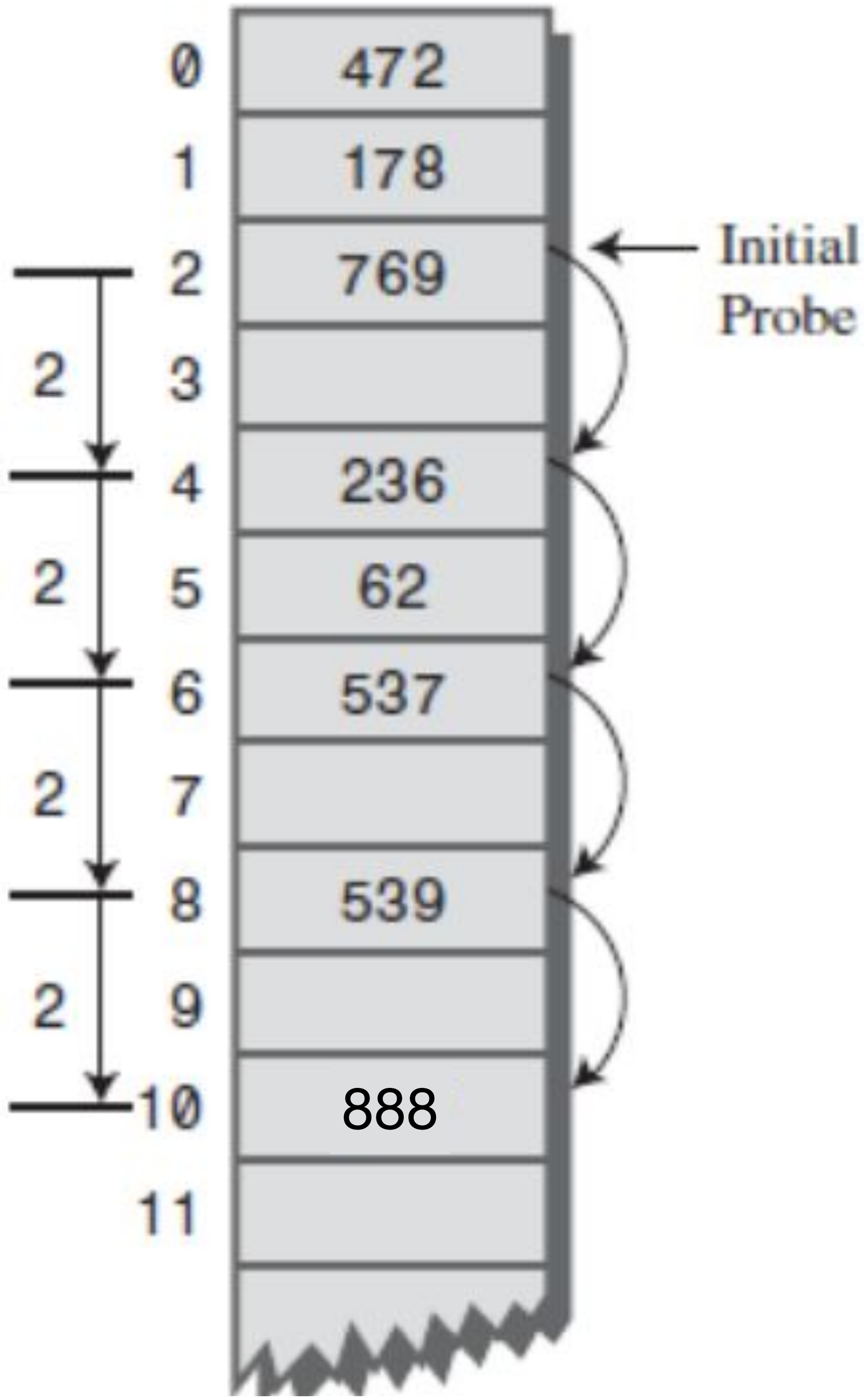
- A good choice for secondary hashing:

```
probe_length = constant - (key % constant);
```

- What's the range of possible values?
- For example:

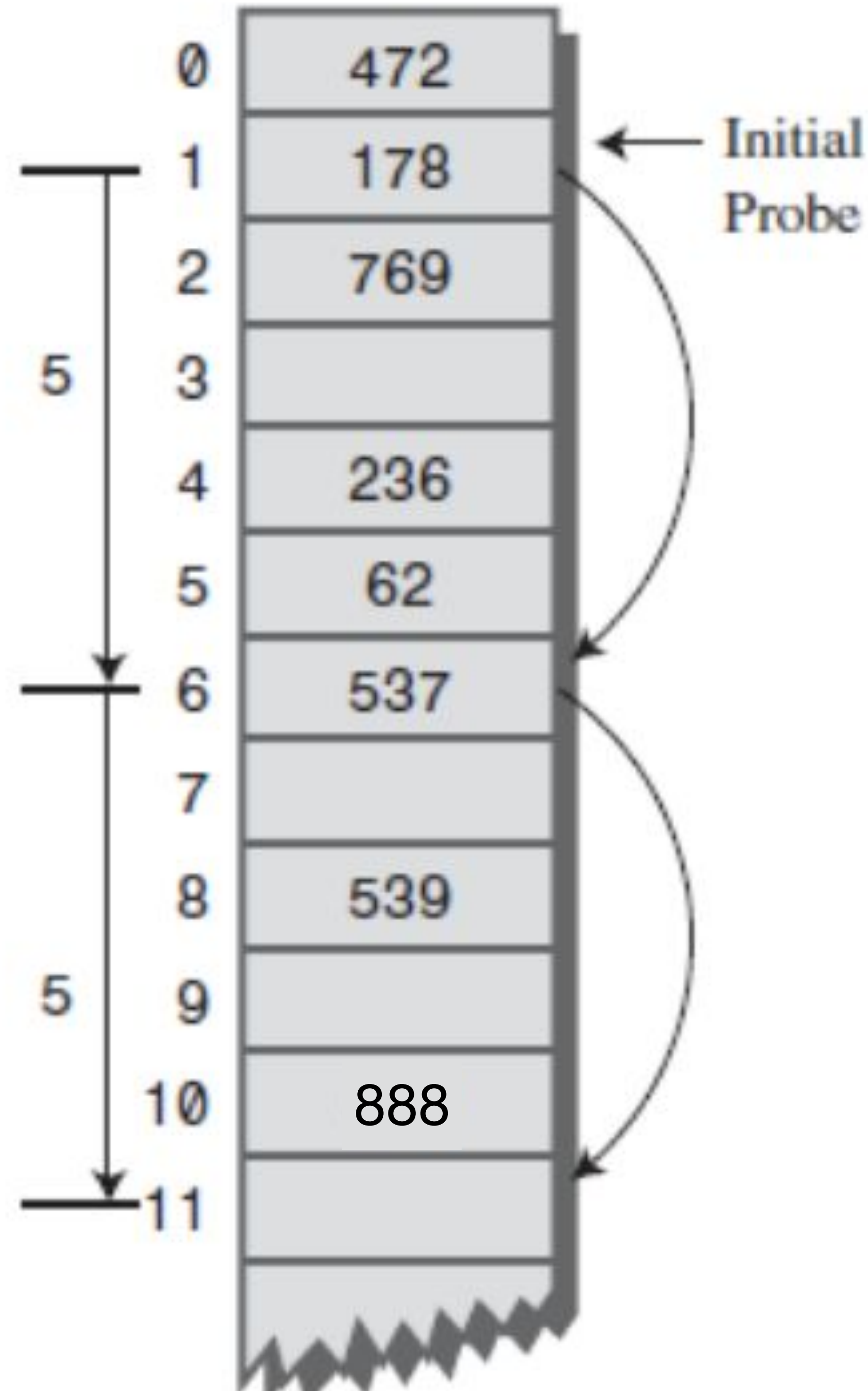
```
probe_length = 5 - (key % 5);
```

- Thus when different keys hash to the same index, they will most likely generate different probe lengths
- Note that this secondary hashing function never generates a probe len of 0. Why is this important?



Key=888

Probe length = [5-888%5]=2



Key=710

Probe length = [5-710%5]=5

Clicker #3

The following hashtable has a **capacity of 10**. The index of each slot is shown below for convenience. Using **double hashing** where the probe length is calculated as `[7-(key%7)]`, where would a new element **80** be inserted into?

60			53	13	25		27		48
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

A. [0]

B. [1]

C. [2]

D. [6]

E. [8]

Load Factor and Rehashing

- **Load Factor**: number of elements divided by the hash table size / capacity.
  - Example: if a table has a capacity of 73, currently storing 40 elements. The load factor is:
    40 / 73 = 56%
- When load factor becomes too large (close to 1, i.e. table getting full), it's necessary to increase the hash table capacity. This is called **re-hashing**.

Clicker #4

```
void rehashing() {
    T[] nheap = (T[]) new Object[heap.length * 2];
    for (int i=0; i < heap.length; i++) {
        Nheap[i] = heap[i];
    }
    heap = nheap;
}
```

What's wrong of this rehashing method?

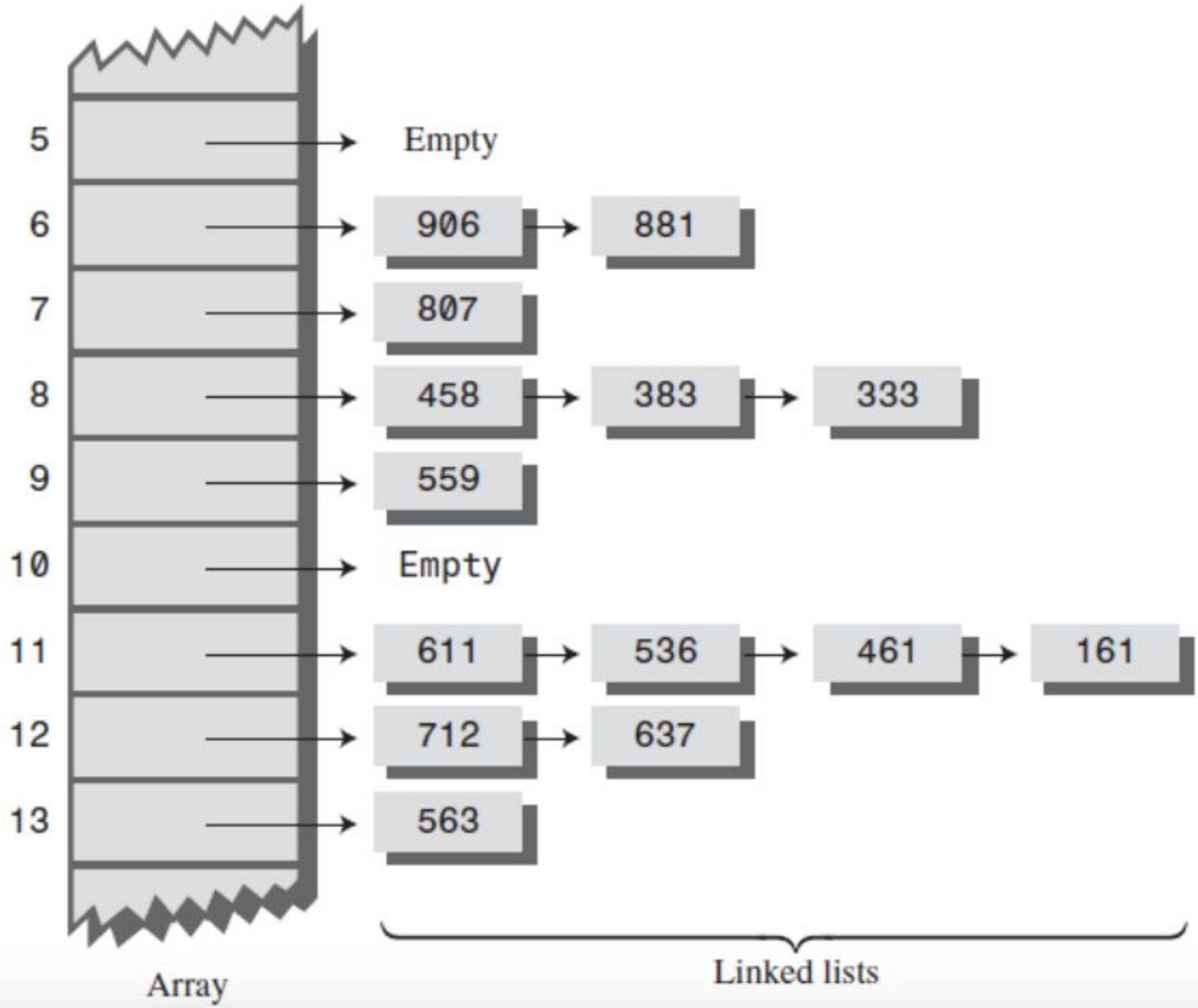
1) It works just fine.
2) It doesn't enlarge the heap space.
3) It doesn't copy the elements to the new heap.
4) It doesn't update hash value of the elements for new heap

Load Factor and Rehashing

- **Load Factor**: number of elements divided by the hash table size / capacity.
  - Example: if a table has a capacity of 73, currently storing 40 elements. The load factor is: 40 / 73 = 56%
- When load factor becomes too large (close to 1, i.e. table getting full), it's necessary to increase the hash table capacity. This is called **re-hashing**.
- Since the table size has changed, you need to rehash every element to its new location in the new hash table. You can't simply copy elements over. Why?

Separate Chaining

- Open Addressing (e.g. linear, quadratic probing and double hashing) resolves collision by looking for an empty slot in the hash table.
- Another idea is to create a linked list at each slot to allow for multiple elements. This is called **Separate Chaining**.
- In separate chaining, deletion is easier because all collided elements go into a linked list, so deletion simply deletes the element from that linked list. No ‘flag’ business any more.


5
6
7
8
9
10
11
12
13
Empty
906
881
807
458
383
333
559
Empty
611
536
461
161
712
637
563
Array
Linked lists

Java’s Hashtable class

- Java provides a `class Hashtable<K, V>` where K is key’s type, V is the value’s type
- It implements the `Map<K, V>` interface, and can re-hash dynamically based on a pre-defined load factor (e.g. 0.75)
- For example:

```
Hashtable<String, Float> table =
    new Hashtable<String, Float>();
```

```
table.put(“John Smith”, 6.0f);
table.put(“Eric May”, 5.8f);
table.put(“Rose Ann”, 5.9f);
System.out.println(table.get(“Rose Ann”));
```

Questions?