

# Open Addressing

261217 Data Structures for Computer Engineers

Patiwet Wuttisarnwattana, Ph.D.

[patiwet@eng.cmu.ac.th](mailto:patiwet@eng.cmu.ac.th)

Computer Engineering, Chiang Mai University

# Chaining is unpredictable

Chained hash tables require special memory allocation

- Can we create a hash table without significant memory allocation?

Explicitly storing references to the next object due to a collision requires memory

- Linked lists require a pointer to the next node

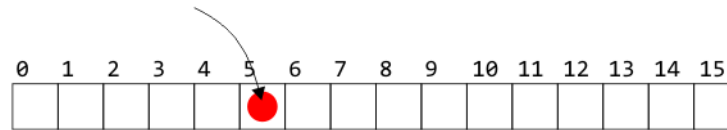
We will deal with collisions by storing collisions elsewhere

- We will define an implicit rule which tells us where to look next

# Open Addressing

Suppose an object hashes to bin 5

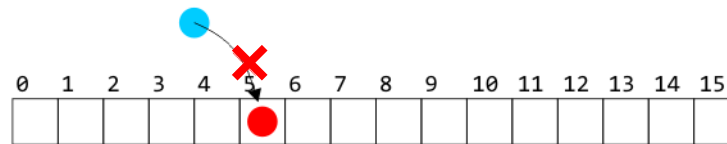
- If bin 5 is empty, we can copy the object into that entry



# Open Addressing

Suppose, however, another object hashes to bin 5

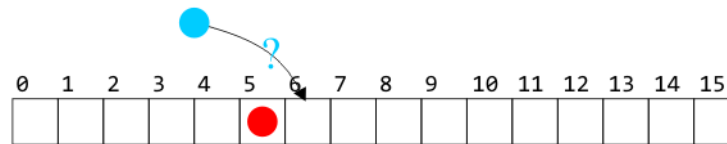
- Without a linked list, we cannot store the object in that bin



# Open Addressing

We could have a rule which says:

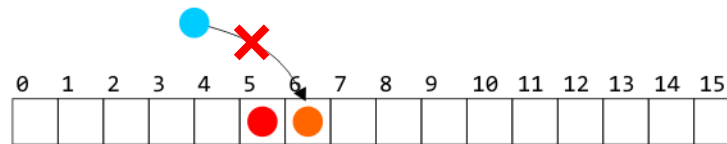
- Look in the next bin to see if it is occupied
- Such a rule is *implicit*—we do not follow an explicit link



# Open Addressing

The rule must be general enough to deal with the fact that the next cell could also be occupied

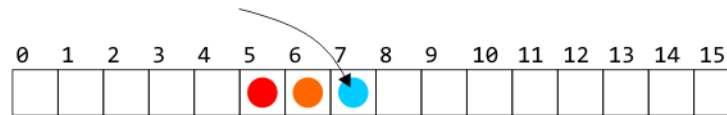
- For example, continue searching until the first empty bin is found
- The rule must be simple to follow—*i.e.*, fast



# Open Addressing

We could then store the object in the next location

- Problem: we can only store as many objects as there are entries in the array: the load factor  $\lambda \leq 1$



# Open Addressing

Of course, whatever rule we use in placing an object must also be used when searching for or removing objects





# Open Addressing

Recall, however, that our goal is  $\Theta(1)$  access times

- We cannot, on average, be forced to access too many bins



# Open Addressing

This short topic introduces the concept of open addressing

- Use a predefined sequence of bins which should be searched
- We need a fast rule that can be easily followed
- We must ensure that we are not making too many searches
- The load factor will never be greater than one

# Linear Probing

Our first scheme for open addressing:

- Linear probing—keep looking ahead one cell at a time
- Examples and implementations
- Primary clustering
- Is it working looking ahead every  $k$  entries?

# Linear Probing

The easiest method to probe the bins of the hash table is to search forward linearly

Assume we are inserting into bin  $k$ :

- If bin  $k$  is empty, we occupy it
- Otherwise, check bin  $k + 1$ ,  $k + 2$ , and so on, until an empty bin is found
  - If we reach the end of the array, we start at the front (bin 0)

# Linear Probing

Consider a hash table with  $M = 16$  bins

Given a 3-digit hexadecimal number:

- The least-significant digit is the primary hash function (bin)
- (The least-significant four bits)
- Example: for  $6B72A_{16}$ , the has value is **A**

# Insertion

Insert these numbers into this initially empty hash table:

19A, 207, 3AD, 488, 5BA, 680, 74C, 826, 946, ACD, B32, C8B, D59, E9C

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

# Insertion

Start with the first four values:

19A, 207, 3AD, 488

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

# Example

Start with the first four values:

19A, 207, 3AD, 488

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
							207	488		19A			3AD		



# Example

Next we must insert 5BA

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
							207	488		19A			3AD		

# Example

Next we must insert 5B**A**

- Bin **A** is occupied
- We search forward for the next empty bin

0	1	2	3	4	5	6	7	8	9	<b>A</b>	B	C	D	E	F
							207	488		19A	<b>5BA</b>		3AD		

# Example

Next we are adding 680, 74C, 826

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
							207	488		19A	5BA		3AD		

# Example

Next we are adding 680, 74C, 826

- All the bins are empty—simply insert them

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680						826	207	488		19A	5BA	74C	3AD		

# Example

Next, we must insert 946

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680						826	207	488		19A	5BA	74C	3AD		

# Example

Next, we must insert 94**6**

- Bin **6** is occupied
- The next empty bin is 9

0	1	2	3	4	5	<b>6</b>	7	8	9	A	B	C	D	E	F
680						826	207	488	<b>946</b>	19A	5BA	74C	3AD		

# Example

Next, we must insert ACD

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680						826	207	488	946	19A	5BA	74C	3AD		

# Example

Next, we must insert ACD

- Bin **D** is occupied
- The next empty bin is E

0	1	2	3	4	5	6	7	8	9	A	B	C	<b>D</b>	E	F
680						826	207	488	946	19A	5BA	74C	3AD	<b>ACD</b>	



# Example

Next, we insert B32

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680						826	207	488	946	19A	5BA	74C	3AD	ACD	

# Example

Next, we insert B3**2**

- Bin **2** is unoccupied

0	1	<b>2</b>	3	4	5	6	7	8	9	A	B	C	D	E	F
680		<b>B32</b>				826	207	488	946	19A	5BA	74C	3AD	ACD	

# Example

Next, we insert C8B

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680		B32				826	207	488	946	19A	5BA	74C	3AD	ACD	

# Example

Next, we insert C8**B**

- Bin **B** is occupied
- The next empty bin is F

0	1	2	3	4	5	6	7	8	9	A	<b>B</b>	C	D	E	F
680		B32				826	207	488	946	19A	5BA	74C	3AD	ACD	<b>C8B</b>

# Example

Next, we insert D59

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680		B32				826	207	488	946	19A	5BA	74C	3AD	ACD	C8B

# Example

Next, we insert D59

- Bin **9** is occupied
- The next empty bin is 1

0	1	2	3	4	5	6	7	8	<b>9</b>	A	B	C	D	E	F
680	<b>D59</b>	B32				826	207	488	946	19A	5BA	74C	3AD	ACD	C8B

# Example

Finally, insert E9C

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32				826	207	488	946	19A	5BA	74C	3AD	ACD	C8B

# Example

Finally, insert E9C

- Bin **C** is occupied
- The next empty bin is 3

0	1	2	3	4	5	6	7	8	9	A	B	<b>C</b>	D	E	F
680	D59	B32	<b>E9C</b>			826	207	488	946	19A	5BA	74C	3AD	ACD	C8B



# Example

Having completed these insertions:

- The load factor is  $\lambda = 14/16 = 0.875$
- The average number of probes is  $38/14 \approx 2.71$

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	3AD	ACD	C8B

# Resizing the array

To double the capacity of the array, each value must be rehashed

- 680, B32, ACD, 5BA, 826, 207, 488, D59 may be immediately placed
  - We use the least-significant five bits for the initial bin

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
680						826	207	488					ACD					B32							D59	5BA					

# Resizing the array

To double the capacity of the array, each value must be rehashed

- 19A resulted in a collision

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
680						826	207	488					ACD					B32							D59	5BA	19A				

# Resizing the array

To double the capacity of the array, each value must be rehashed

- 946 resulted in a collision

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	
680							826	207	488	946				ACD					B32							D59	5BA	19A				

# Resizing the array

To double the capacity of the array, each value must be rehashed

- 74C fits into its bin

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
680						826	207	488	946			74C	ACD				946	B32							D59	5BA	19A				

# Resizing the array

To double the capacity of the array, each value must be rehashed

- 3AD resulted in a collision

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
680						826	207	488	946			74C	ACD	3AD			946	B32							D59	5BA	19A				

# Resizing the array

To double the capacity of the array, each value must be rehashed

- Both E9C and C8B fit without a collision
- The load factor is  $\lambda = 14/32 = 0.4375$
- The average number of probes is  $18/14 \approx 1.29$

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
680						826	207	488	946		C8B	74C	ACD	3AD			946	B32							D59	5BA	19A	E9C			

# Searching

Testing for membership is similar to insertions:

Start at the appropriate bin, and searching forward until

1. The item is found,
2. An empty bin is found, or
3. We have traversed the entire array

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	3AD	ACD	C8B

The third case will only occur if the hash table is full (load factor of 1)



# Searching

Searching for C8B

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	3AD	ACD	C8B

# Searching

Searching for C8**B**

- Examine bins B, C, D, E, F
- The value is found in Bin F

0	1	2	3	4	5	6	7	8	9	A	<b>B</b>	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	3AD	ACD	<b>C8B</b>

# Searching

Searching for 23E

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	3AD	ACD	C8B

# Searching

Searching for 23E

- Search bins E, F, 0, 1, 2, 3, 4
- The last bin is empty; therefore, 23E is not in the table

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93	×		826	207	488	946	19A	5BA	74C	3AD	ACD	C8B

# Erasing

We cannot simply remove elements from the hash table

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	3AD	ACD	C8B

# Erasing

We cannot simply remove elements from the hash table

- For example, consider erasing 3AD

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	3AD	ACD	C8B

# Erasing

We cannot simply remove elements from the hash table

- For example, consider erasing 3AD
- If we just erase it, it is now an empty bin
  - By our algorithm, we cannot find ACD, C8B and D59

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C		ACD	C8B

# Erasing

Instead, we must attempt to fill the empty bin

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C		ACD	C8B



# Erasing

Instead, we must attempt to fill the empty bin

- We can move ACD into the location

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	<del>ACB</del>	ACD	C8B

# Erasing

Now we have another bin to fill

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	ACD		C8B

# Erasing

Now we have another bin to fill

- We can move ACD into the location

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	ACD	<del>C8B</del>	C8B

# Erasing

Now we must attempt to fill the bin at F

- We cannot move 680

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	ACD	C8B	

# Erasing

Now we must attempt to fill the bin at F

- We cannot move 680
- We can, however, move D59

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	ACD	C8B	<b>D59</b>

# Erasing

At this point, we cannot move B32 or E93 and the next bin is empty

- We are finished

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680		B32	E93			826	207	488	946	19A	5BA	74C	ACD	C8B	D59

# Erasing

Suppose we delete 207

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680		B32	E93			826	207	488	946	19A	5BA	74C	ACD	C8B	D59

# Erasing

Suppose we delete 207

- Cannot move 488

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680		B32	E93			826		488	946	19A	5BA	74C	ACD	C8B	D59



# Erasing

Suppose we delete 207

- We could move 946 into Bin 7

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680		B32	E93			826	946	488	946	19A	5BA	74C	ACD	C8B	D59

# Erasing

Suppose we delete 207

- We cannot move either the next five entries

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680		B32	E93			826	946	488		19A	5BA	74C	ACD	C8B	D59

# Erasing

Suppose we delete 207

- We cannot move either the next five entries

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680		B32	E93			826	946	488	<b>D59</b>	<del>19A</del>	<del>5BA</del>	<del>74C</del>	<del>ACD</del>	<del>C8B</del>	<del>D59</del>

# Erasing

Suppose we delete 207

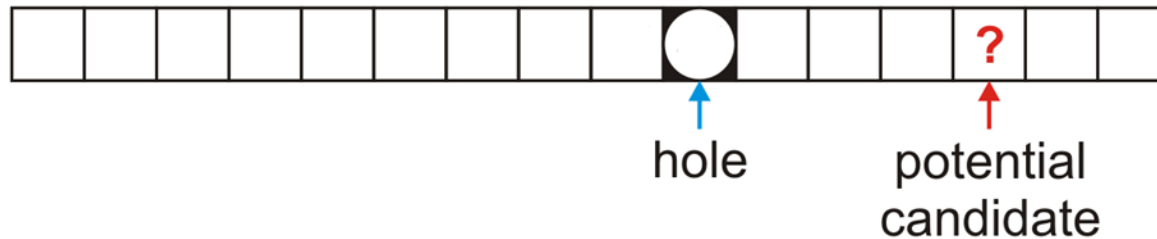
- We cannot fill this bin with 680, and the next bin is empty
- We are finished

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680		B32	E93			826	946	488	D59	19A	5BA	74C	ACD	C8B	

# Erasing

In general, assume:

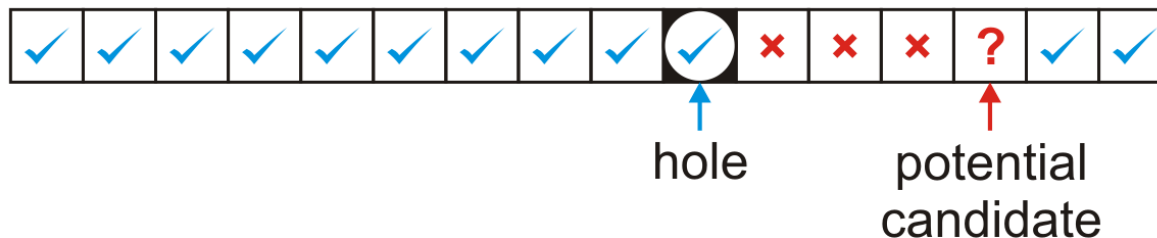
- The currently removed object has created a hole at index **hole**
- The object we are checking is located at the position **index** and has a hash value of hash



# Erasing

The first possibility is that  $\text{index} > \text{hole}$

- Move the object if the candidate has  
(the hash value  $\leq$  the hole) AND (the hash value  $<$  index)



- Remember: if we are checking the object ? at location index, this means that all entries between hole and index are both occupied and could not have been copied into the hole

# Erasing

The other possibility is we wrapped around the end of the array, that is,  $\text{index} < \text{hole}$

- Move the object if the candidate has  
(the hash value  $\leq \text{hole}$ ) AND (the hash value  $> \text{index}$ )



In either case, if the move is successful, the ? Now becomes the new hole to be filled

# Primary Clustering

We have already observed the following phenomenon:

- With more insertions, the contiguous regions (or *clusters*) get larger

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
680						826	207	488	946		C8B	74C	ACD	3AD			946	B32							D59	5BA	19A	E9C			


This results in longer search times



# Primary Clustering

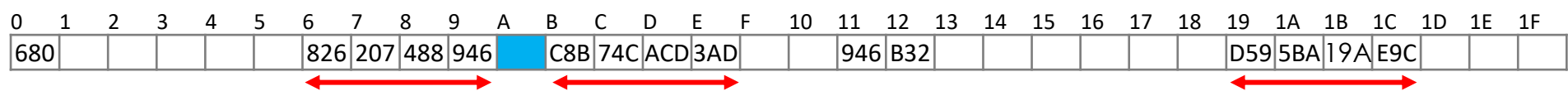
We currently have three clusters of length four

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	
680						826	207	488	946		C8B	74C	ACD	3AD			946	B32								D59	5BA	19A	E9C			



# Primary Clustering

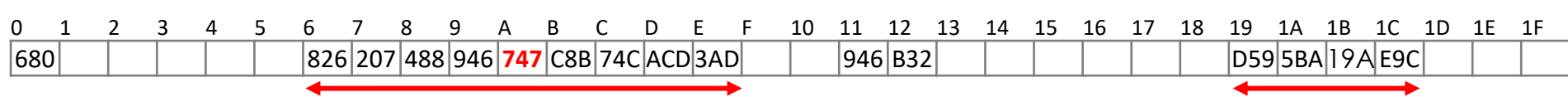
There is a  $5/32 \approx 16\%$  chance that an insertion will fill Bin A



# Primary Clustering

There is a  $5/32 \approx 16\%$  chance that an insertion will fill Bin A


- This causes two clusters to *coalesce* into one larger cluster of length 9



# Primary Clustering

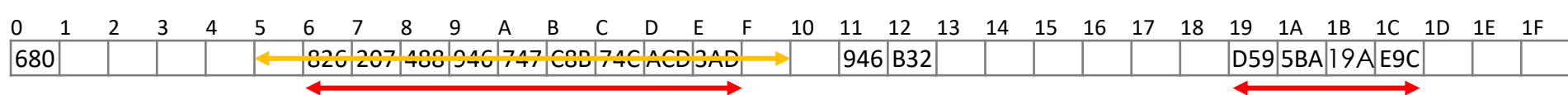
There is now a  $11/32 \approx 34\%$  chance that the next insertion will increase the length of this cluster

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
680						826	207	488	946	747	C8B	74C	ACD	3AD			946	B32							D59	5BA	19A	E9C			



# Primary Clustering

As the cluster length increases, the probability of further increasing the length increases



In general:

- Suppose that a cluster is of length  $\ell$
- An insertion either into any bin occupied by the chain or into the locations immediately before or after it will increase the length of the chain
- This gives a probability of  $\frac{\ell + 2}{M}$

# Run-time analysis

The length of these chains will affect the number of probes required to perform insertions, accesses, or removals

It is possible to estimate the average number of probes for a successful search, where  $\lambda$  is the load factor:

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

For example: if  $\lambda = 0.5$ , we require 1.5 probes on average

# Run-time analysis

The number of probes for an unsuccessful search or for an insertion is higher:

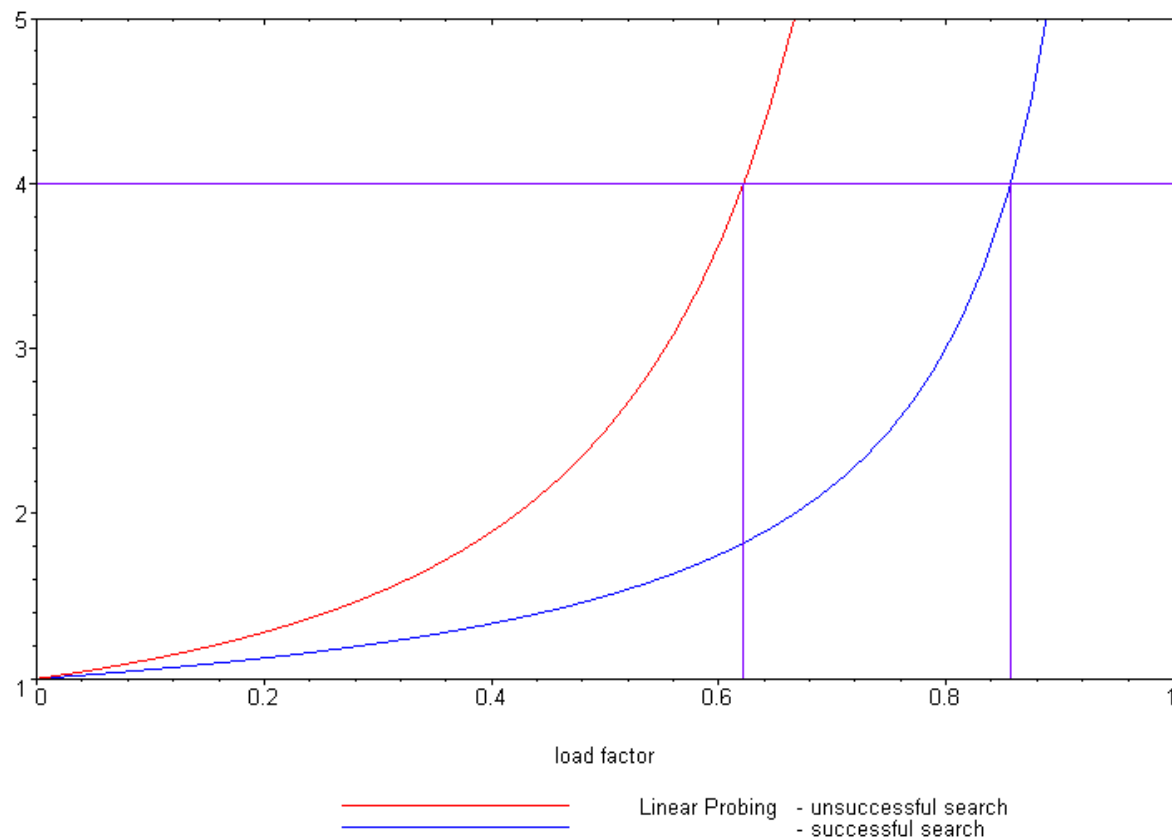
$$\frac{1}{2} \left( 1 + \frac{1}{(1 - \lambda)^2} \right)$$

For  $0 \leq \lambda \leq 1$ , then  $(1 - \lambda)^2 \leq 1 - \lambda$ , and therefore the reciprocal will be larger

- Again, if  $\lambda = 0.5$  then we require 2.5 probes on average

# Run-time analysis

The following plot shows how the number of required probes increases





# Run-time analysis

Our goal was to keep all operations  $\Theta(1)$

Unfortunate, as  $\lambda$  grows, so does the run time

One solution is to keep the load factor under a given bound

If we choose  $\lambda = 2/3$ , then the number of probes for either a successful or unsuccessful search is 2 and 5, respectively

# Run-time analysis

Therefore, we have three choices:

- Choose  $M$  large enough so that we will not pass this load factor
  - This could waste memory
- Double the number of bins if the chosen load factor is reached
  - Not available if dynamic memory allocation is not available
- Choose a different strategy from linear probing
  - Two possibilities are quadratic probing and double hashing

# Summary

This topic introduced linear problem

- Continue looking forward until an empty cell is found
- Searching follows the same rule
- Removing an object is more difficult
- Primary clustering is an issue
- Keep the load factor  $\lambda \leq 2/3$

# Quadratic Probing

This topic covers quadratic probing

- Similar to linear probing
  - Does not step forward one step at a time
- Primary clustering no longer occurs
- Affected by secondary clustering

Linear probing:

- Look at bins  $k, k + 1, k + 2, k + 3, k + 4, \dots$
- Primary clustering

# Quadratic Probing

Linear probing causes primary clustering

- All entries follow the same search pattern for bins:

```
int initial = hash_M( x, M );  
for ( int k = 0; k < M; ++k ) {  
    bin = (initial + k) % M;  
    // ...  
}
```



# Quadratic Probing

Quadratic probing suggests moving forward by different amounts

For example,

```
int initial = hash_M( x, M );  
  
for ( int k = 0; k < M; ++k ) {  
    bin = (initial + k*k) % M;  
}
```

# Quadratic Probing

Problem:

- Will  $\text{initial} + k*k$  step through all of the bins?
- Here, the array size is 10:

```
M = 10;  
initial = 5
```

```
for ( int k = 0; k <= M; ++k ) {  
    std::cout << (initial + k*k) % M << ' ';  
}
```

- The output is

5 6 9 4 **1** 0 **1** 4 9 6 5

# Quadratic Probing

## Problem:

- Will  $\text{initial} + k*k$  step through all of the bins?
- Now the array size is 12:

```
M = 12;  
initial = 5
```

```
for ( int k = 0; k <= M; ++k ) {  
    std::cout << (initial + k*k) % M << ' ' ;  
}
```

- The output is now

5 6 9 2 9 6 5 6 9 2 9 6 5



# Making $M$ Prime

If we make the table size  $M = p$  a prime number quadratic probing is guaranteed to iterate through  $\left\lceil \frac{p}{2} \right\rceil$  entries

Problems:

- All operations must be done using %
  - Cannot use &, <<, or >>
  - The modulus operator % is relatively slow
- Doubling the number of bins is difficult:
  - What is the next prime after  $2 \times 263$ ?

# Generalization

More generally, we could consider an approach like:

```
int initial = hash_M( x, M );  
  
for ( int k = 0; k < M; ++k ) {  
    bin = (initial + c1*k + c2*k*k) % M;  
}
```

# Using $M = 2^m$

If we ensure  $M = 2^m$  then choose

$$c_1 = c_2 = 1/2$$

```
int initial = hash_M( x, M );  
  
for ( int k = 0; k < M; ++k ) {  
    bin = (initial + (k + k*k)/2) % M;  
}
```

- Note that  $k + k*k$  is always even
- The growth is still  $\Theta(k^2)$
- This guarantees that all  $M$  entries are visited before the pattern repeats
  - This only works for powers of two

# Using $M = 2^m$

For example:

- Use an array size of 16:

```
M = 16;
```

```
initial = 5
```

```
for ( int k = 0; k <= M; ++k ) {  
    std::cout << (initial + (k + k*k)/2) % M << ' '  
}
```

- The output is now

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

# Using $M = 2^m$

There is an even easier means of calculating this approach

```
int bin = hash_M( x, M );  
  
for ( int k = 0; k < M; ++k ) {  
    bin = (bin + k) % M;  
}
```

- Recall that  $M = 2^m$ , so just keep adding the next highest value

# Example

Consider a hash table with  $M = 16$  bins

Given a 2-digit hexadecimal number:

- The least-significant digit is the primary hash function (bin)
- Example: for  $6B7A_{16}$ , the initial bin is **A**

# Example

Insert these numbers into this initially empty hash table

9A, 07, AD, 88, BA, 80, 4C, 26, 46, C9, 32, 7A, BF, 9C

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

# Example

Start with the first four values:

9A, 07, AD, 88

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F



# Example

Start with the first four values:

9A, 07, AD, 88

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
							07	88		9A			AD		

# Example

Next we must insert BA

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
							07	88		9A			AD		

# Example

Next we must insert BA

- The next bin is empty

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
							07	88		9A	BA		AD		

# Example

Next we are adding 80, 4C, 26

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
							07	88		9A	BA		AD		

# Example

Next we are adding 80, 4C, 26

- All the bins are empty—simply insert them

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80						26	07	88		9A	BA	4C	AD		

# Example

Next, we must insert 46

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80						26	07	88		9A	BA	4C	AD		

# Example

Next, we must insert 46

- Bin **6** is occupied
- Bin **6 + 1 = 7** is occupied
- Bin **7 + 2 = 9** is empty

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80						26	07	88	46	9A	BA	4C	AD		

# Example

Next, we must insert C9

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80						26	07	88	46	9A	BA	4C	AD		



# Example

Next, we must insert C9

- Bin **9** is occupied
- Bin **9 + 1 = A** is occupied
- Bin **A + 2 = C** is occupied
- Bin **C + 3 = F** is empty

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80						26	07	88	46	9A	BA	4C	AD		C9

# Example

Next, we insert 32

- Bin 2 is unoccupied

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80		32				26	07	88	46	9A	BA	4C	AD		C9

# Example

Next, we insert 7A

- Bin **A** is occupied
- Bins **A + 1 = B**, **B + 2 = D** and **D + 3 = 0** are occupied
- Bin **0 + 4 = 4** is empty

0	1	2	3	4	5	6	7	8	9	<b>A</b>	B	C	D	E	F
80		32		<b>7A</b>		26	07	88	46	9A	BA	4C	AD		C9

# Example

Next, we insert BF

- Bin **F** is occupied
- Bins **F + 1 = 0** and **0 + 2 = 2** are occupied
- Bin **2 + 3 = 5** is empty

0	1	2	3	4	5	6	7	8	9	<b>A</b>	B	C	D	E	F
80		32		7A	<b>BF</b>	26	07	88	46	9A	BA	4C	AD		C9

# Example

Finally, we insert 9C

- Bin **C** is occupied
- Bins **C + 1 = D**, **D + 2 = F**, **F + 3 = 2**, **2 + 4 = 6** and **6 + 5 = B** are occupied
- Bin **B + 6 = 1** is empty

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80	9C	32		7A	BF	26	07	88	46	9A	BA	4C	AD		C9

# Example

Having completed these insertions:

- The load factor is  $\lambda = 14/16 = 0.875$
- The average number of probes is  $32/14 \approx 2.29$

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80	9C	32		7A	BF	26	07	88	46	9A	BA	4C	AD		C9

■ HW ends here

# Resizing the array

To double the capacity of the array, each value must be rehashed

- 80, 9C, 32, 7A, BF, 26, 07, 88 may be immediately placed
  - We use the least-significant five bits for the initial bin

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
80						26	07	88										32								7A		9C			BF

- If the next least-significant digit is
  - Even, use bins 0 – F
  - Odd, use bins 10 – 1F



# Resizing the array

To double the capacity of the array, each value must be rehashed

- 46 results in a collision
  - We place it in bin 9

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
80						26	07	88	46									32								7A		9C			BF

# Resizing the array

To double the capacity of the array, each value must be rehashed

- 9A results in a collision
  - We place it in bin 1B

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
80						26	07	88	46									32								7A	9A	9C			BF

# Resizing the array

To double the capacity of the array, each value must be rehashed

- BA also results in a collision
  - We place it in bin 1D

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
80						26	07	88	46									32								7A	9A	9C	BA		BF

# Resizing the array

To double the capacity of the array, each value must be rehashed

- 4C and AD don't cause collisions

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
80						26	07	88	46			4C	AD					32								7A	9A	9C	BA		BF

# Resizing the array

To double the capacity of the array, each value must be rehashed

- Finally, C9 causes a collision
  - We place it in bin A

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
80						26	07	88	46	C9		4C	AD					32								7A	9A	9C	BA		BF

# Resizing the array

To double the capacity of the array, each value must be rehashed

- The load factor is  $\lambda = 14/32 = 0.4375$
- The average number of probes is  $20/14 \approx 1.43$

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
80						26	07	88	46	C9		4C	AD					32								7A	9A	9C	BA		BF

# Erase

Can we erase an object like we did with linear probing?

- Consider erasing 9A from this table
- There are  $M - 1$  possible locations where an object which could have occupied a position could be located

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80	21		43			76				9A					50

Instead, we will use the concept of *lazy deletion*

- Mark a bin as ERASED; however, when searching, treat the bin as occupied and continue
  - We must have a separate ternary-valued flag for each bin

# Erase

If we erase AD, we must mark that bin as erased

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80	9C	32		7A	BF	26	07	88	46	9A	BA	4C	<del>AD</del>		C9



# Find

When searching, it is necessary to skip over this bin

- For example, find AD: D, E  
find 5C: C, D, F, 2, 5, 9, F, 6, E

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80	9C	32		7A	BF	26	07	88	46	9A	BA	4C	<del>AD</del>		C9

# Modified insertion

We must modify insert, as we may place new items into either

- Unoccupied bins
- Erased bins

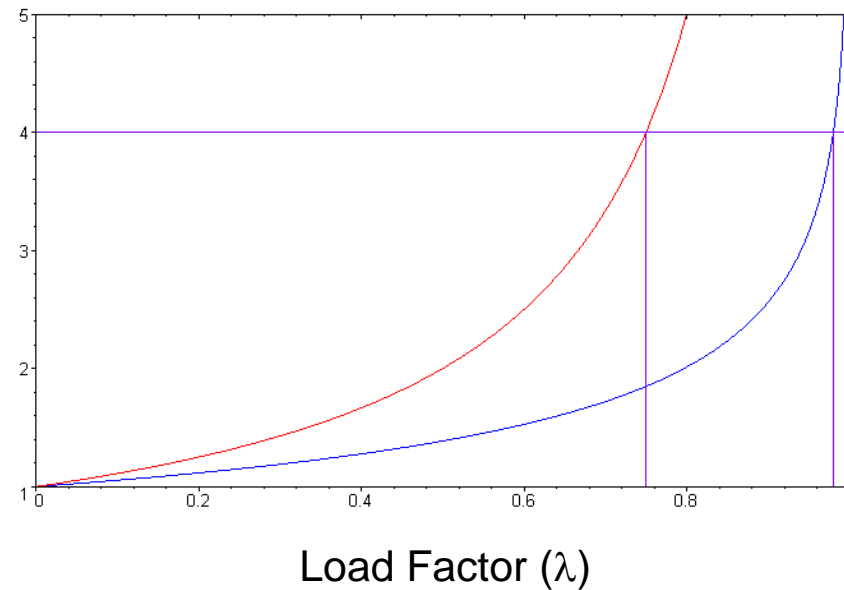
# Expected number of probes

It is possible to calculate the expected number of probes for quadratic probing, again, based on the load factor:

- Successful searches:  $\ln\left(\frac{1}{1-\lambda}\right) / \lambda$
- Unsuccessful searches:  $\frac{1}{1-\lambda}$

When  $\lambda = 2/3$ , we requires 1.65 and 3 probes, respectively

- Linear probing required 3 and 5 probes, respectively



— Unsuccessful search  
— Successful search

# Quadratic probing versus linear probing

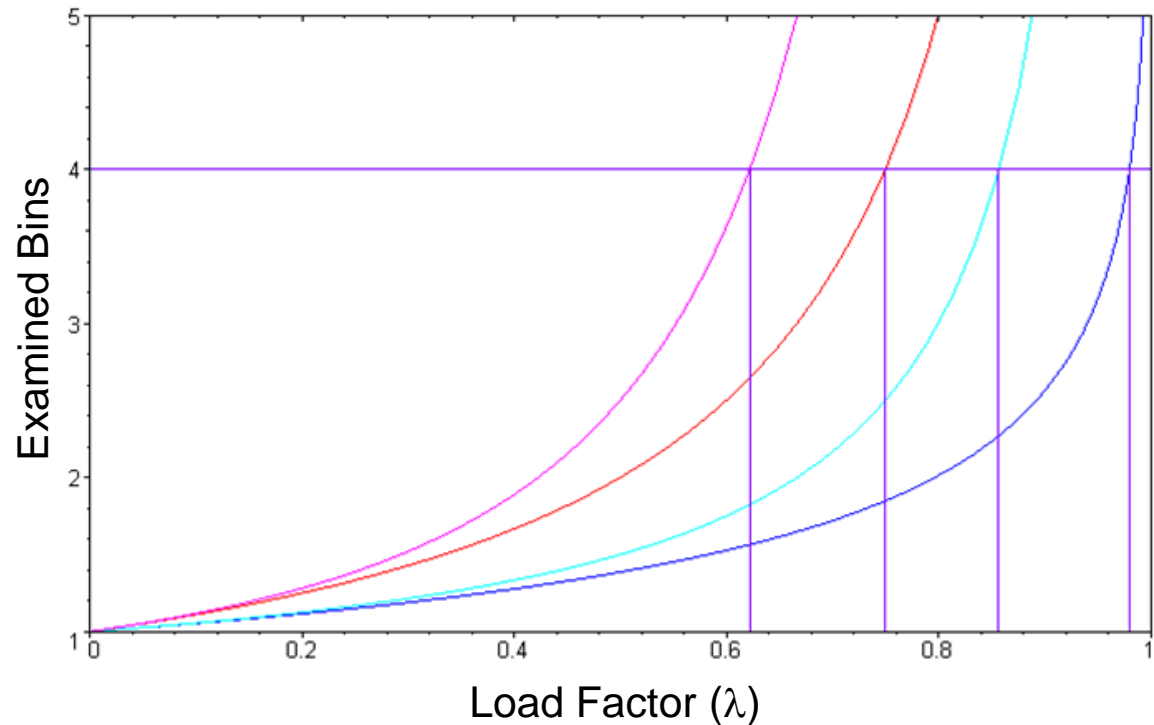
Comparing the two:

## Linear probing

Unsuccessful search  
Successful search

## Quadratic probing

Unsuccessful search  
Successful search



# Cache misses

One benefit of quadratic probing:

- The first few bins examined are close to the initial bin
- It is unlikely to reference a section of the array far from the initial bin

Modern computers use caches

- 4 KiB *pages* of main memory are copied into faster caches
- Pages are only brought into the cache when referenced
- Accesses close to the initial bin are likely to reference the same page

# Summary

In this topic, we have looked at quadratic probing:

- An open addressing technique
- Steps forward by a quadratically growing steps
- Insertions and searching are straight forward
- Removing objects is more complicated: use lazy deletion
- Still subject to secondary probing