Open addressing

Outline

Chained hash tables require special memory allocation

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

We will deal with collisions by storing collisions elsewhere

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

Background

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

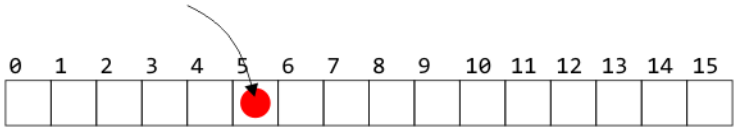
- Linked lists require a pointer to the next node
- Scatter tables require a pointer to the next cell

We will implicit rules which dictate where to look next

Open Addressing

Suppose an object hashes to bin 5

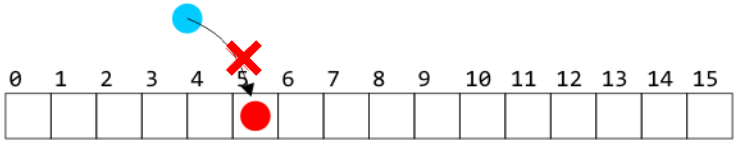
- If bin 5 is empty, we can copy the object into that entry



Open Addressing

Suppose, however, another object hashes to bin 5

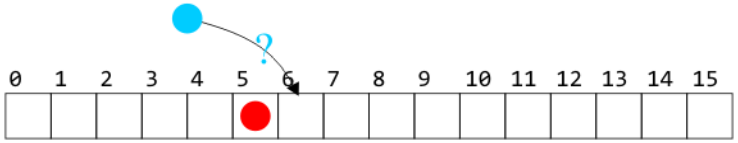
- Without a linked list, we cannot store the object in that bin



Open Addressing

We could have a rule which says:

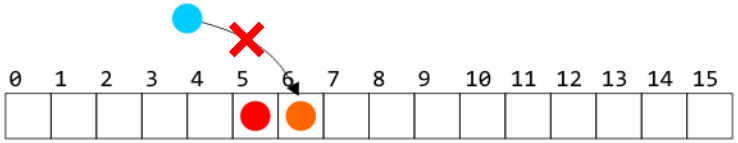
- Look in the next bin to see if it is occupied
- Such a rule is *implicit*—we do not follow an explicit link



Open Addressing

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

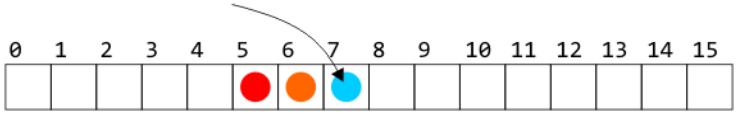
- For example, continue searching until the first empty bin is found
- The rule must be simple to follow—*i.e.*, fast



Open Addressing

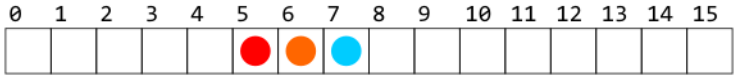
We could then store the object in the next location

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



Open Addressing

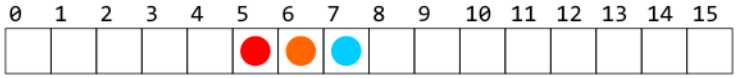
Of course, whatever rule we use in placing an object must also be used when searching for or removing objects



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

There are numerous strategies for defining the order in which the bins should be searched:

- Linear probing
- Quadratic probing
- Double hashing

There are many alternate strategies, as well:

- Last come, first served
  - Always place the object into the bin moving what may be there already
- Cuckoo hashing

Summary

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

References

Wikipedia, http://en.wikipedia.org/wiki/Hash_function

[1] Cormen, Leiserson, and Rivest, *Introduction to Algorithms*, McGraw Hill, 1990.

[2] Weiss, Data Structures and Algorithm Analysis in C++, 3rd Ed., Addison Wesley.

Linear probing

Outline

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

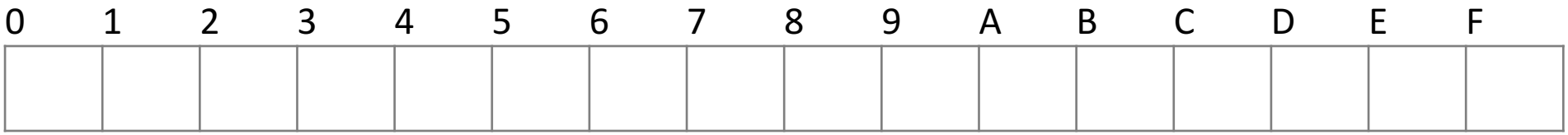
Given a 3-digit hexadecimal number:

- The least-significant digit is the primary hash function (bin)
- Example: for $6B72A_{16}$, the initial bin is **A** and the jump size is **3**

Insertion

Insert these numbers into this initially empty hash table:

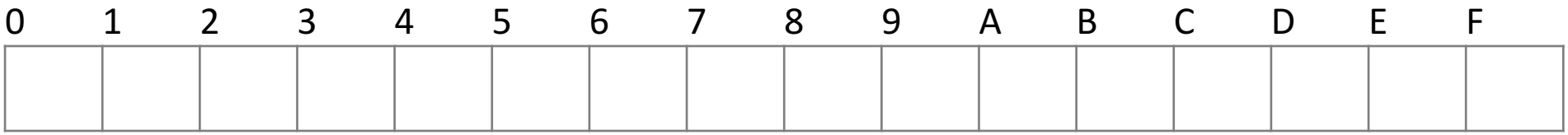
19A, 207, 3AD, 488, 5BA, 680, 74C, 826, 946, ACD, B32, C8B, DBE, E9C



Example

Start with the first four values:

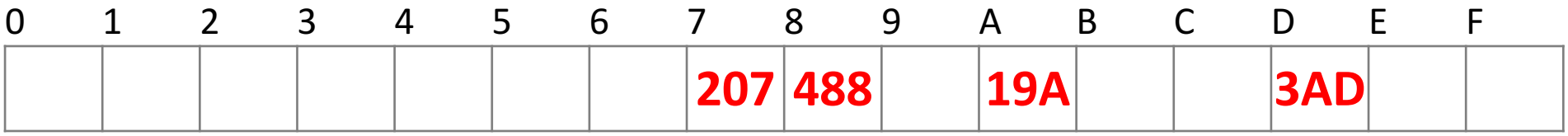
19A, 207, 3AD, 488



Example

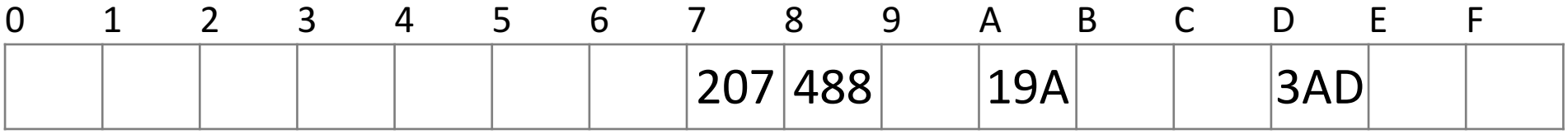
Start with the first four values:

19**A**, 20**7**, 3A**D**, 48**8**



Example

Next we must insert 5BA



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	**A**	B	C	D	E	F
							207	488		19A	**5BA**		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 68**0**, 74**C**, 82**6**

- 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 946

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

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

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	D	E	F
680						826	207	488	946	19A	5BA	74C	3AD	ACD	

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 B32

- Bin 2 is unoccupied

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 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**	C	D	E	F
680		B32				826	207	488	946	19A	5BA	74C	3AD	ACD	**C8B**

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	**9**	A	B	C	D	E	F
680	**D59**	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	C	D	E	F
680	D59	B32	**E9C**			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			

Marking bins occupied

How can we mark a bin as occupied?

Pointers	`nullptr`
Positive integers	`-1`
Floating-point numbers	`NaN`
Objects	Create a privately stored static object that does not compare to any other instances of that class

Suppose we're storing arbitrary integers?

- Should we store `-1938275734` in the hopes that it will never be inserted into the hash table?
- In general, *magic numbers* are bad—they lead to spurious errors

A better solution:

- Create a bit vector where the k^{th} entry is marked true if the k^{th} entry of the hash table is occupied

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**	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	3AD	ACD	**C8B**

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

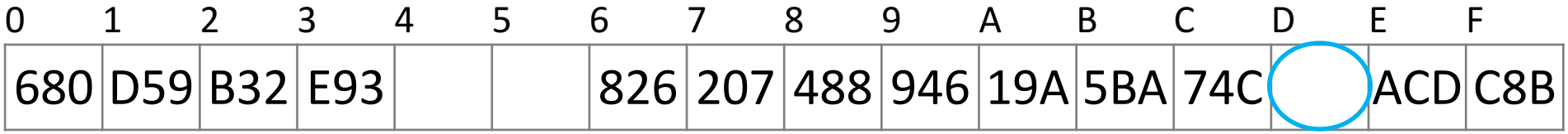
We cannot simply remove elements from the hash table

- 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

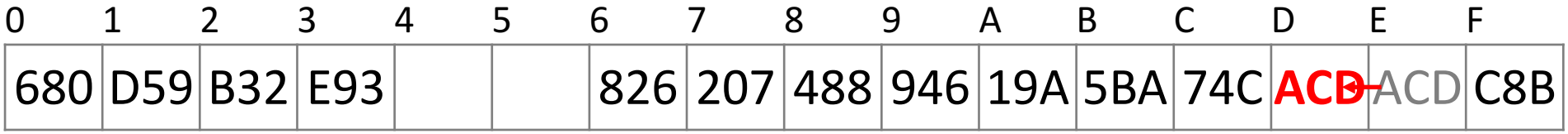
Instead, we must attempt to fill the empty bin



Erasing

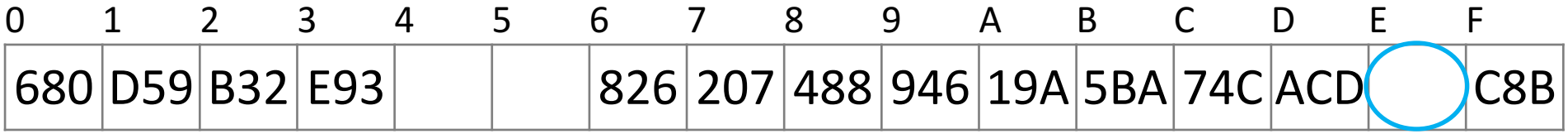
Instead, we must attempt to fill the empty bin

- We can move ACD into the location



Erasing

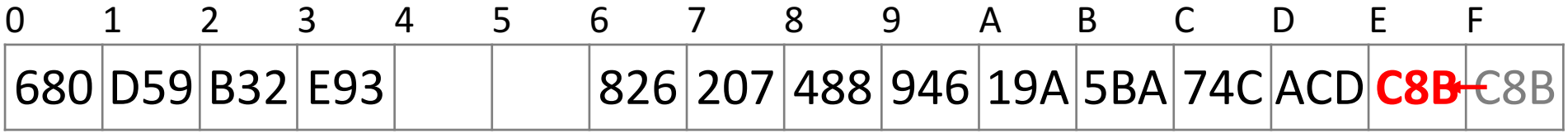
Now we have another bin to fill



Erasing

Now we have another bin to fill

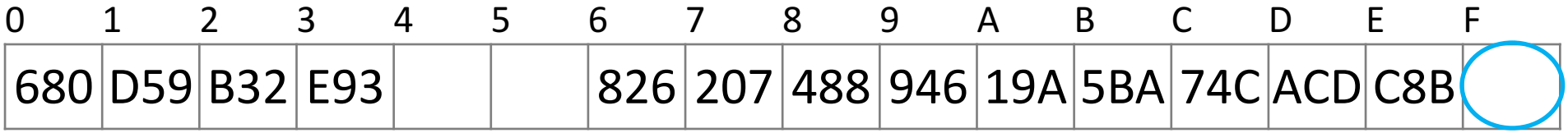
- We can move ACD into the location



Erasing

Now we must attempt to fill the bin at F

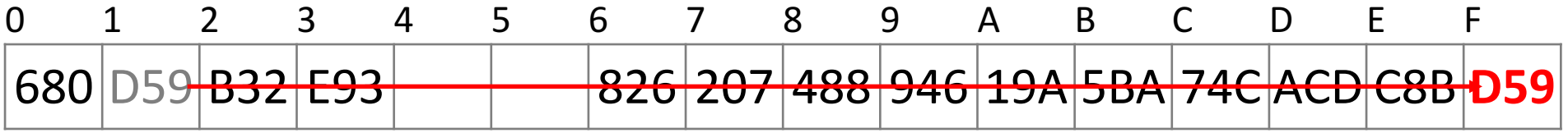
- We cannot move 680



Erasing

Now we must attempt to fill the bin at F

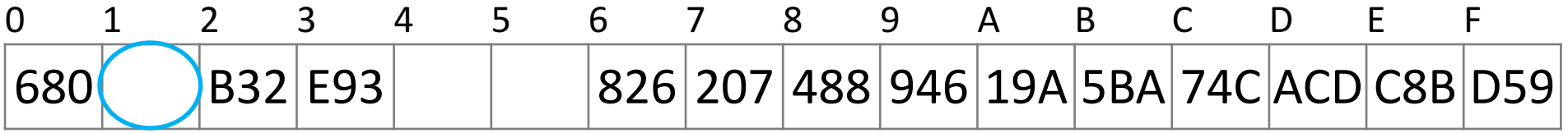
- We cannot move 680
- We can, however, move D59



Erasing

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

- We are finished



Erasing

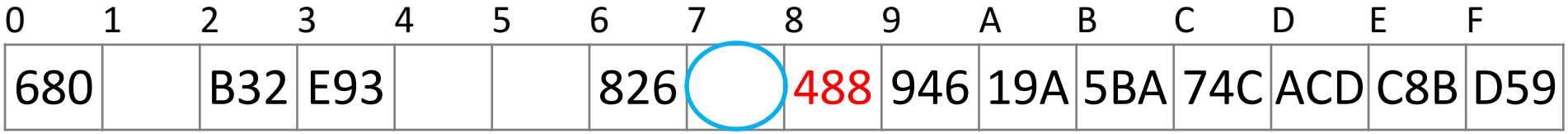
Suppose we delete 207

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

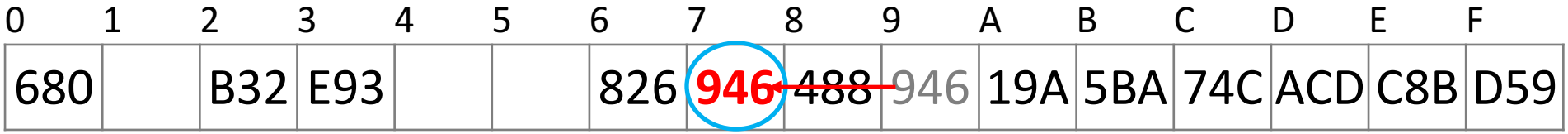
- Cannot move 488



Erasing

Suppose we delete 207

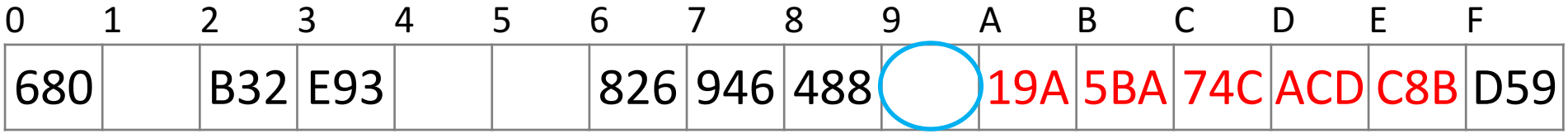
- We could move 946 into Bin 7



Erasing

Suppose we delete 207

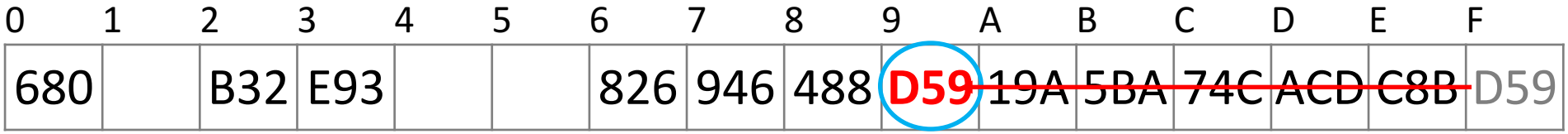
- We cannot move either the next five entries



Erasing

Suppose we delete 207

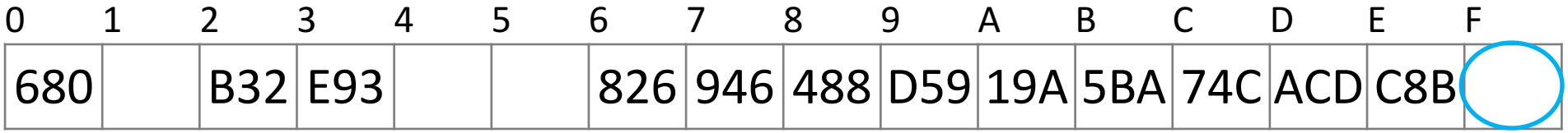
- We cannot move either the next five entries



Erasing

Suppose we delete 207

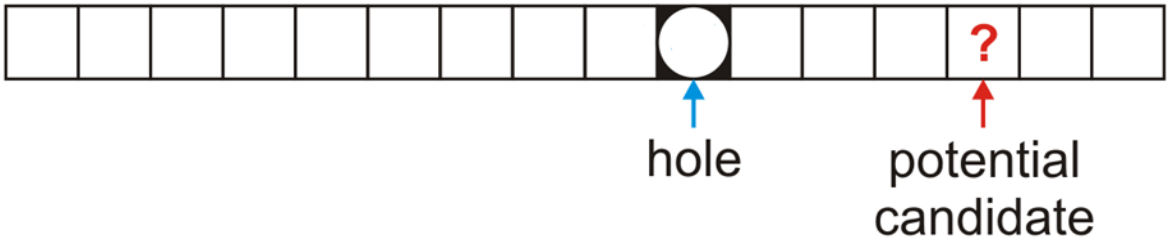
- We cannot fill this bin with 680, and the next bin is empty
- We are finished



Erasing

In general, assume:

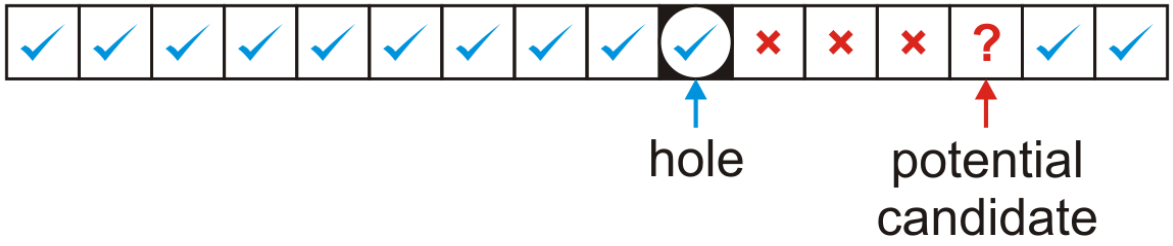
- The currently removed object has created a hole at index `hole`
- The object we are checking is located at the position `index` and has a hash value of `hash`



Erasing

The first possibility is that `hole < index`

- In this case, the hash value of the object at `index` must either
  - equal to or less than the `hole` **or**
  - it must be greater than the `index` of the potential candidate

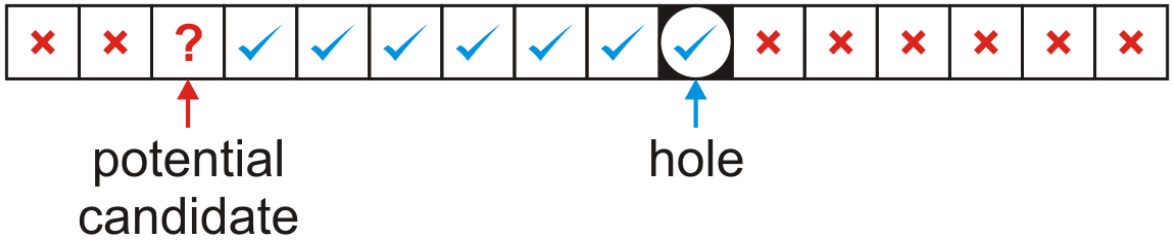


- Remember: if we are checking the object `?` at location `index`, this means that all entries between `hole` and `index` are both occupied and could not have been copied into the hole

Erasing

The other possibility is we wrapped around the end of the array, that is, `hole > index`

- In this case, the hash value of the object at `index` must be both
  - greater than the index of the potential candidate **and**
  - it must be less than or equal to the hole



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

Black Board Example

Using the last digit as our hash function—insert these nine numbers into a hash table of size $M = 10$

$$31, 15, 79, 55, 42, 99, 60, 80, 23$$

Then, remove 79, 31, 42, and 60, in that order

Primary Clustering

We have already observed the following phenomenon:

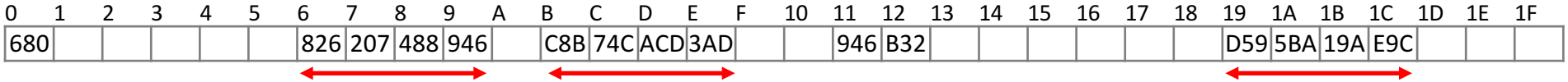
- With more insertions, the contiguous regions (or *clusters*) get larger

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

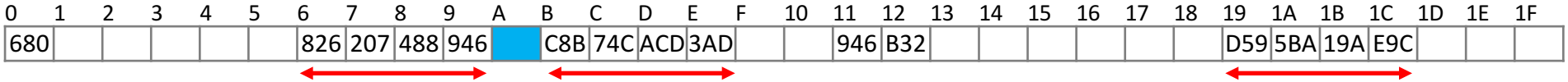
Primary Clustering

We currently have three clusters of length four



Primary Clustering

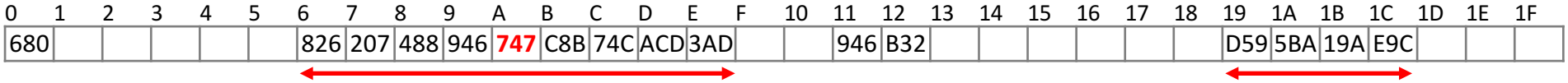
There is a $5/32 \approx 16\ \%$ chance that an insertion will fill Bin A



Primary Clustering

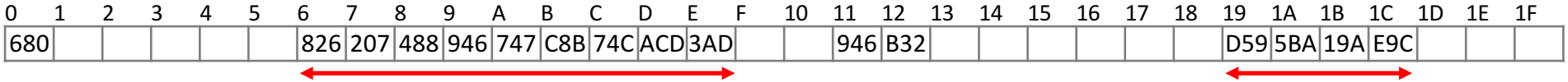
There is a $5/32 \approx 16\ \%$ chance that an insertion will fill Bin A

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



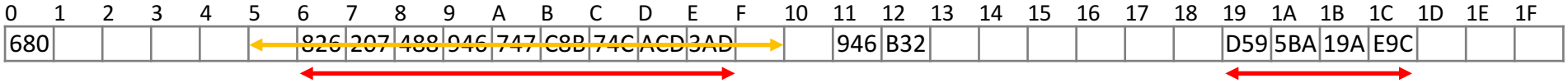
Primary Clustering

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



Primary Clustering

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



In general:

- Suppose that a cluster is of length ℓ
- 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 $\frac{l+2}{M}$

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 $\lambda \leq 2/3$

References

Wikipedia, http://en.wikipedia.org/wiki/Hash_function

[1] Cormen, Leiserson, and Rivest, *Introduction to Algorithms*, McGraw Hill, 1990.

[2] Weiss, Data Structures and Algorithm Analysis in C++, 3rd Ed., Addison Wesley.

Double hashing

Outline

This topic covers double hashing

- More complex than linear or quadratic probing
- Uses two hash functions
  - The first gives the bin
  - The second gives the jump size
- Primary clustering no longer occurs
- More efficient than linear or quadratic probing

Background

Linear probing:

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

Quadratic probing:

- Look at bins $k, k+1, k+4, k+9, k+16, \ldots$
- Secondary clustering (dangerous for poor hash functions)
- Expensive:
  - Prime-sized arrays
  - Euclidean algorithm for calculating remainders

Background

Linear probing causes ***primary clustering***

– All entries follow the same search pattern for bins:

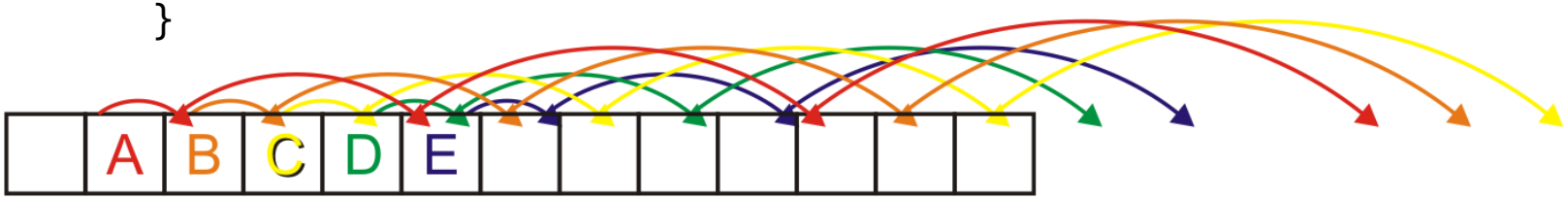
```
int initial = hash_M( x.hash(), M );
for ( int k = 0; k < M; ++k ) {
    bin = (initial + k) % M;
    // ...
}
```

Background

This can be partially solved with quadratic probing

– The step size grows in quadratic increments: 0, 1, 4, 9, 16, ...

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



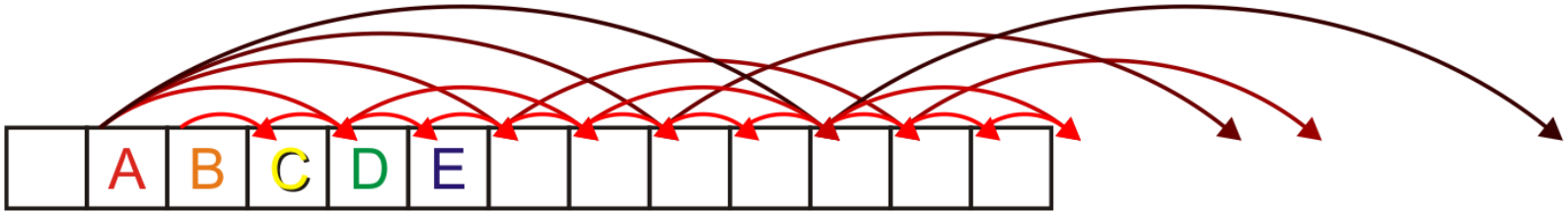
– Problems: what happens if multiple things are placed in the same bin?

- The same steps are followed for each: ***secondary clustering***
- This indicates a potentially poor hash function
- This sometimes cannot be avoided

Background

An alternate solution?

– Give each object (with high probability) a different jump size



Description

Associate with each object an initial bin and a jump size

For example,

```
int initial = hash_M( x.hash(), M );
int jump    = hash2_M( x.hash(), M );

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

Description

Problem:

- Will `initial + k*jump` step through all of the bins?
- Here, the jump size is 7:

```
M = 16;
initial = 5
jump    = 7;

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

- The output is

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

Description

Problem:

- Will `initial + k*jump` step through all of the bins?
- Now, the jump size is 12:

```
M = 16;
initial = 5
jump    = 12;

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

- The output is now

```
5 1 13 9 5 1 13 9 5 1 13 9 5 1 13 9 5
```

Description

The jump size and the number of bins M must be *coprime*[1]

- They must share no common factors

There are two ways of ensuring this:

- Make M a prime number
- Restrict the prime factors

[1] also known as *relatively prime*

Making M Prime

If we make the table size $M = p$ a prime number then all values between $1, ..., p - 1$ are relatively prime

- Example: 13 does not share any common factors with $1, 2, 3, ..., 11, 12$

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 × 263?

Using $M = 2^m$

We can restrict the number of prime factors

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

- The only prime factor of M is 2
- All odd numbers are relatively prime to M

Benefits:

- Doubling the size of the array is easy
- We can use bitwise operations

Using $M = 2^m$

Given a number, how do we ensure the jump size is odd?

Make the least-significant bit a 1:

```
unsigned int make_odd( unsigned int n ) {
    return n | 1;
}
```

For example:

```
00101011011001001111000?0101010?
                    |
00000000000000000000000000000001
                    =
00101011011001001111000101010101
```

Using $M = 2^m$

This also works for signed integers and 2's complement:

- The least significant bit of a negative odd number is 1

For example

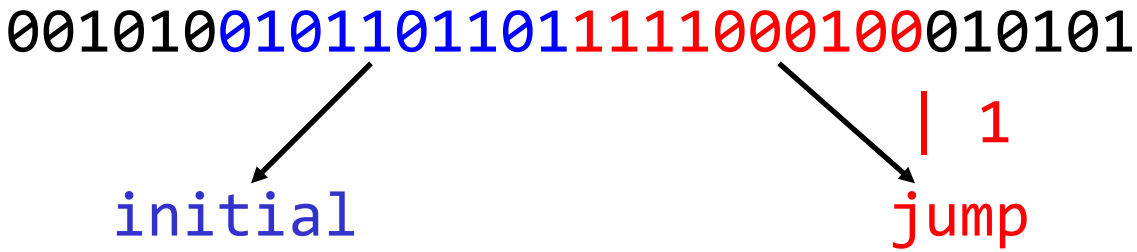
–1	11111111111111111111111111111111
–2	11111111111111111111111111111110
–3	11111111111111111111111111111101
–4	11111111111111111111111111111100
–5	11111111111111111111111111111011
–6	11111111111111111111111111111010
–7	11111111111111111111111111111001
–8	11111111111111111111111111111000
–9	11111111111111111111111111110111

Using $M = 2^m$

Devising a second hash function is necessary

One solution: define two functions `hash_M1` and `hash_M2` which map onto $0, \ldots, M-1$

- Use a different set of m bits, *e.g.*, if $m = 10$ use



where `initial == 365` and `jump == 965`

Useful only if $m \leq 16$

Using $M = 2^m$

Another solution:

```
int initial = hash_M( x.hash(),                M );
int jump    = hash_M( x.hash() * 532934959, M ) | 1;

for ( int k = 0; k < M; ++k ) {
    bin = (initial + k*jump) & (M - 1);
    // ...
}
```

532934959 is prime

Example

Consider a hash table with $M = 16$ bins

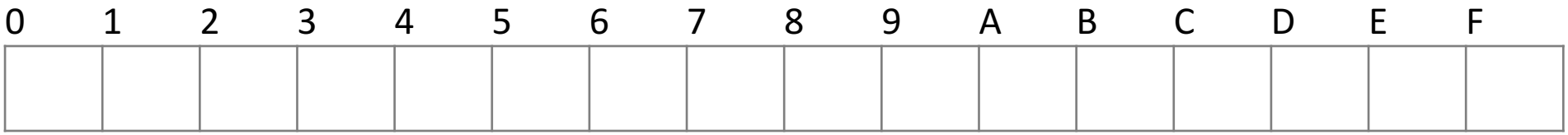
Given a 3-digit hexadecimal number:

- The least-significant digit is the primary hash function (bin)
- The next digit is the secondary hash function (jump size)
- Example: for $6B72A_{16}$, the initial bin is **A** and the jump size is **3**

Example

Insert these numbers into this initially empty hash table

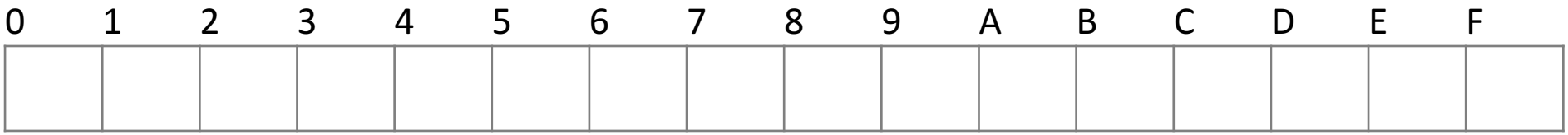
19A, 207, 3AD, 488, 5BA, 680, 74C, 826, 946, ACD, B32, C8B, DBE, E9C



Example

Start with the first four values:

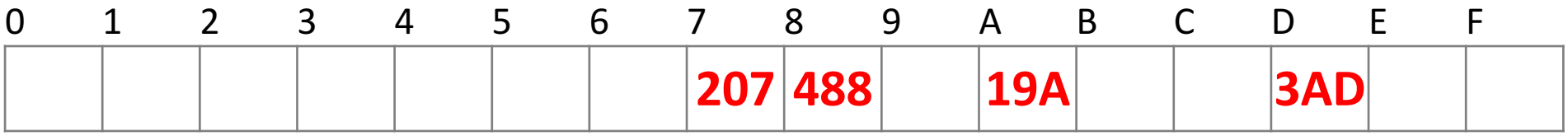
19A, 207, 3AD, 488



Example

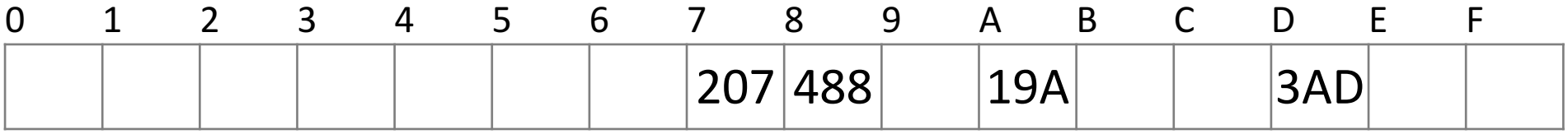
Start with the first four values:

19**A**, 20**7**, 3A**D**, 48**8**



Example

Next we must insert 5BA



Example

Next we must insert 5BA

- Bin A is occupied
- The jump size is B is already odd
- A jump size of B is equal to a jump size of B – 10_{16} = –5

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

- Bin A is occupied
- The jump size is B is already odd
- A jump size of B is equal to a jump size of B – 10_{16} = –5

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

- The sequence of bins is A, 5

Example

Next we are adding 680, 74C, 826

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

Example

Next we are adding 68**0**, 74**C**, 82**6**

- 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					5BA	**826**	207	488		19A		**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					5BA	826	207	488		19A		74C	3AD		

Example

Next, we must insert 946

- Bin **6** is occupied
- The second digit is 4, which is even
- The jump size is **4** + 1 = 5

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

Example

Next, we must insert 946

- Bin **6** is occupied
- The second digit is 4, which is even
- The jump size is **4** + 1 = 5

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

- The sequence of bins is 6, B

Example

Next, we must insert ACD

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

Example

Next, we must insert ACD

- Bin D is occupied
- The jump size is C is even, so C + 1 = D is odd
- A jump size of D is equal to a jump size of D – 10_{16} = –3

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

Example

Next, we must insert ACD

- Bin **D** is occupied
- The jump size is **C** is even, so **C** + 1 = D is odd
- A jump size of D is equal to a jump size of D – 10_{16} = –3

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

- The sequence of bins is D, A, 7, 4

Example

Next, we insert B32

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

Example

Next, we insert B32

– Bin 2 is unoccupied

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

Example

Next, we insert C8B

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

Example

Next, we insert C8B

- Bin **B** is occupied
- The jump size is **8** is even, so 8 + 1 = 9 is odd
- A jump size of 9 is equal to a jump size of $9 - 10_{16} = -7$

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

Example

Next, we insert C8B

- Bin **B** is occupied
- The jump size is 8 is even, so 8 + 1 = 9 is odd
- A jump size of 9 is equal to a jump size of $9 - 10_{16} = -7$

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

- The sequence of bins is B, 4, D, 6, F

Example

Inserting D5**9**, we note that bin 9 is unoccupied

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

Example

Finally, insert E9C

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

Example

Finally, insert E9C

- Bin C is occupied
- The jump size is 9 is odd
- A jump size of 9 – 10_{16} = –7

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

Example

Finally, insert E9C

- Bin C is occupied
- The jump size is 9 is odd
- A jump size of 9 – 10_{16} = –7

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

- The sequence of bins is C, 5, E

Example

Having completed these insertions:

- The load factor is $\lambda = 14/16 = 0.875$
- The average number of probes is $25/14 \approx 1.79$

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680		B32		ACD	5BA	826	207	488	D59	19A	946	74C	3AD	E9C	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
- The jump size is now 00011001101010_2 or C + 1 = D = 13_{10}
  - We are using the next five bits for the jump size

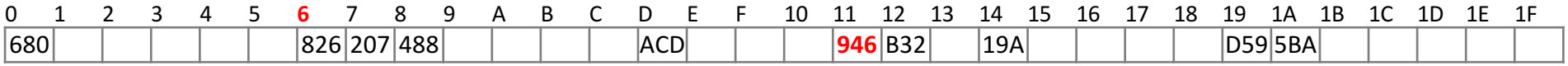
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
680						826	207	488					ACD					B32		19A					D59	5BA					

- The sequence of bins: 1A, 7, 14

Resizing the array

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

- 946 resulted in a collision
- The jump size is now 10010101000110_2 or A + 1 = B = 11_{10}



- The sequence of bins: 6, 11

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				**74C**	ACD				946	B32		19A					D59	5BA					

Resizing the array

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

- 3AD resulted in a collision
- The jump size is now $00111010110 1_2$ or 1D = $29_{10} \equiv_{32} -3_{10}$

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		3AD		74C	ACD				946	B32		19A					D59	5BA					

- The sequence of bins: D, A

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		3AD	C8B	74C	ACD				946	B32		19A					D59	5BA		E9C			

Erase

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

- Clearly, no, as there are $M/2$ possible locations where an object which could have occupied a position could be located

As with quadratic probing, we will use *lazy deletion*

- Mark a bin as `ERASED`; however, when searching, treat the bin as occupied and continue

```
enum bin_state_t {
    UNOCCUPIED,
    OCCUPIED,
    ERASED
};
```

```
bin_state_t state[M];

for ( int i = 0; i < M; ++i ) {
    state[i] = UNOCCUPIED;
}
```

Erase

If we erase 3AD, we must mark that bin as erased

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680		B32		ACD	5BA	826	207	488	959	19A	946	74C	~~3AD~~	E9C	C8B

Find

When searching, it is necessary to skip over this bin

– For example, find ACD: D, A, 7, 4

find C8B: B, 4, D, 6, F

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680		B32		ACD	5BA	826	207	488	959	19A	946	74C	~~3AD~~	E9C	C8B

Multiple insertions and erases

One problem which may occur after multiple insertions and removals is that numerous bins may be marked as `ERASED`

- In calculating the load factor, an `ERASED` bin is equivalent to an `OCCUPIED` bin

This will increase our run times…

Multiple insertions and erases

We can easily track the number of bins which are:

- `UNOCCUPIED`
- `OCCUPIED`
- `ERASED`

by updating appropriate counters

If the load factor λ grows too large, we have two choices:

- If the load factor due to occupied bins is too large, double the table size
- Otherwise, rehash all of the objects currently in the hash table

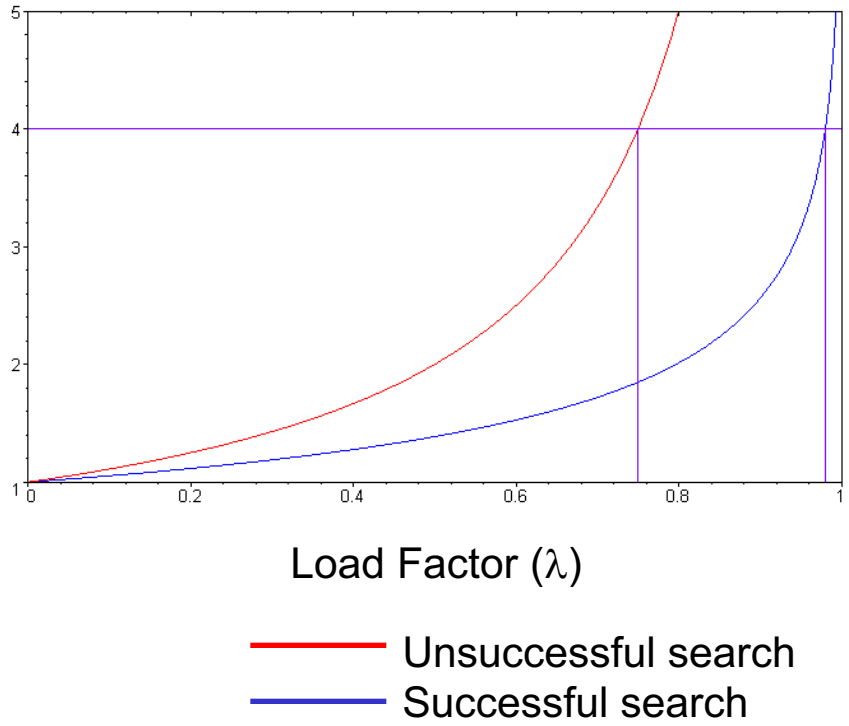
Expected number of probes

As with quadratic probing, the number of probes is significantly lower than for linear probing:

- Successful searches: $\ln \frac{1}{1}$
- Unsuccessful searches: $\frac{1}{1}$

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

- Linear probing required 3 and 5 probes, respectively



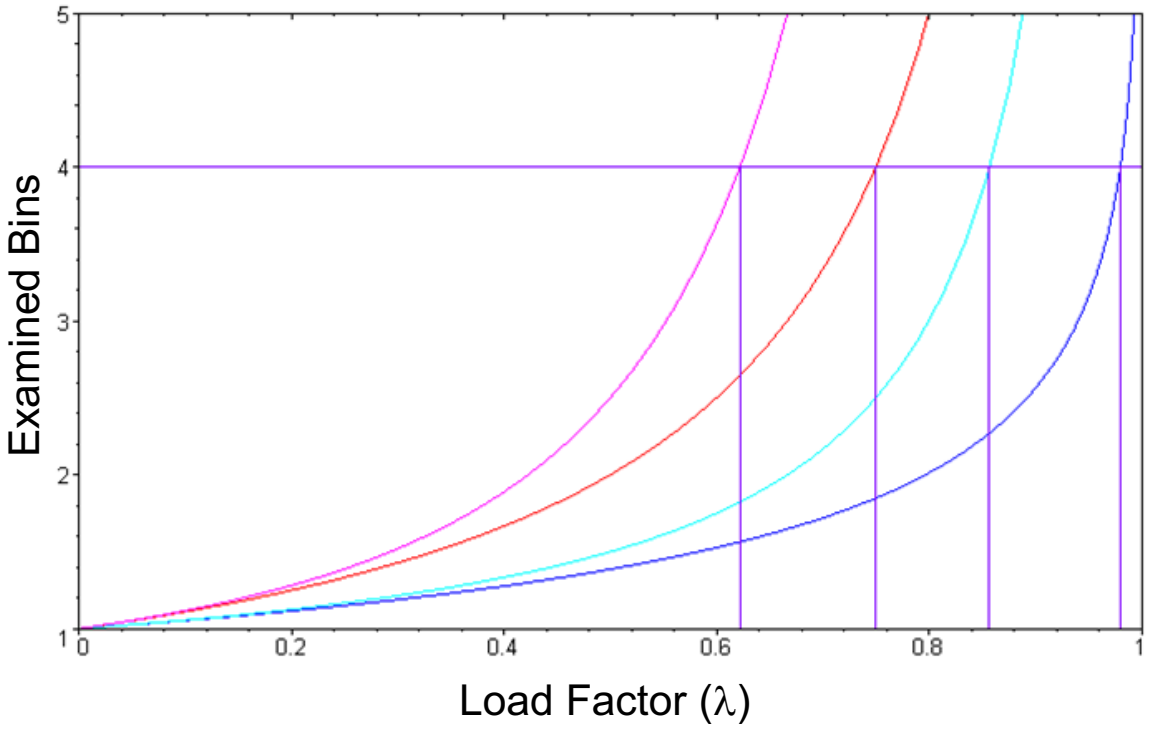
Reference: Knuth, The Art of Computer Programming, Vol. 3, 2nd Ed., 1998, Addison Wesley, p. 530.

Double hashing versus linear probing

Comparing the two:

Linear probing
- Unsuccessful search
- Successful search

Double hashing
- Unsuccessful search
- Successful search



Cache misses

Double hashing solves the secondary clustering problem

- Unfortunately, each subsequent probe could be anywhere in the array
- For large arrays, it is unlikely that block is in the cache
  - This will flag a cache miss and another page will be copied to the cache
- This is slower than quadratic probing
- It may also remove another page from the cache that is to be called again soon in the future
  - If a change was made to the page in the cache, it must be copied out
- When at all possible, use quadratic probing

Summary

In this topic, we have looked at double hashing:

- An open addressing technique
- Uses two hash functions:
  - The first indicates the bin
  - The second gives the jump size
- Insertions and searching are straight forward
- Removing objects is more complicated: use lazy deletion
- It may be useful, on occasion, to clean the table
- Much worse than quadratic probing with respect to cache misses

References

Wikipedia, http://en.wikipedia.org/wiki/Hash_function

[1] Cormen, Leiserson, and Rivest, *Introduction to Algorithms*, McGraw Hill, 1990.

[2] Weiss, Data Structures and Algorithm Analysis in C++, 3rd Ed., Addison Wesley.