Hashing II

Shahriar Ivan
Lecturer, Department of CSE, IUT

Background

- We want to store objects in an array of size M
- We want to quickly calculate the bin where we want to store the object
  - We need to come up with hash function that are hopefully $\Theta(1)$
  - Perfect hash functions (no collisions) are difficult to design
- We will look at some schemes for dealing with collisions

Implementation

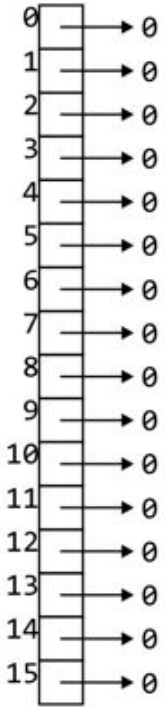
Consider associating each bin with a linked list:

```
template <class Type>
class Chained_hash_table {
    private:
        int table_capacity;
        int table_size;
        Single_list<Type> *table;

        unsigned int hash( Type const & );

    public:
        Chained_hash_table( int = 16 );
        int count( Type const & ) const;
        void insert( Type const & );
        int erase( Type const & );
        // ...
};
```

Implementation

The constructor creates an array of n linked lists

```
template <class Type>
Chained_hash_table::Chained_hash_table( int n ):
table_capacity( std::max( n, 1 ) ),
table_size( 0 ),
table( new Single_list<Type>[table_capacity] ) {
    // empty constructor
}
```



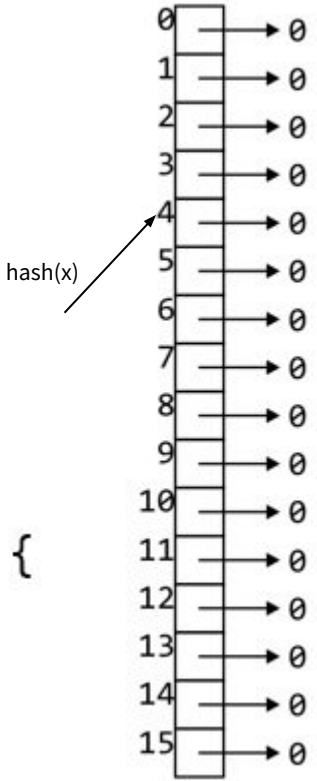
Implementation

The function hash will determine the bin of an object

```
template <class Type>
int Chained_hash_table::hash( Type const &obj ) {
    return hash_M( obj.hash(), capacity() );
}
```

Here, the hash_M() function returns 0, …, M-1:

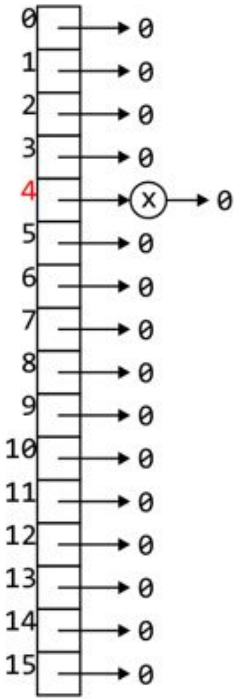
```
unsigned int hash_M( unsigned int n, unsigned int M ) {
    return n % M;
}
```



Implementation

Other functions map to corresponding linked list functions:

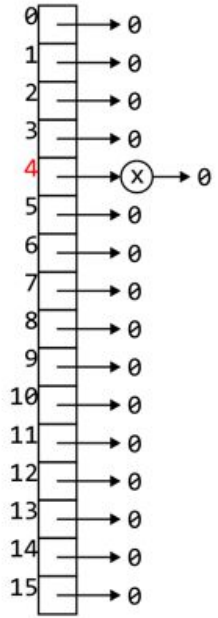
```
template <class Type>
void Chained_hash_table::insert( Type const &obj ) {
    unsigned int bin = hash( obj );

    if ( table[bin].count( obj ) == 0 ) {
        table[bin].push_front( obj );
        ++table_size;
    }
}
```



Implementation

Other functions map to corresponding linked list functions:

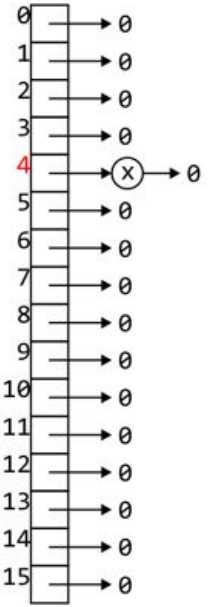
```
template <class Type>
int Chained_hash_table::count( Type const &obj ) const {
    return table[hash( obj )].count( obj );
}
```



Implementation

Other functions map to corresponding linked list functions:

```
template <class Type>
int Chained_hash_table::erase( Type const &obj ) {
    unsigned int bin = hash( obj );

    if ( table[bin].erase( obj ) ) {
        --table_size;
        return 1;
    } else {
        return 0;
    }
}
```



Load Factor

To describe the length of the linked lists, we define the load factor of the hash table:

$$\lambda = \frac{n}{M}$$

This is the average number of objects per bin

- This assumes an even distribution

Load Factor

If the load factor becomes too large, access times will start to increase: O(λ)

The most obvious solution is to double the size of the hash table

- Unfortunately, the hash function must now change

Problems with Linked Lists

- One significant issue with chained hash tables using linked lists
  - It requires extra memory
  - It uses dynamic memory allocation
- For faster access, we could replace each linked list with an AVL tree (assuming we can order the objects)
  - The access time drops to $O(\ln(\lambda))$
  - The memory requirements are increased by $\Theta(n)$, as each node will require two pointers

Open Addressing

Chained hash tables require special memory allocation

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

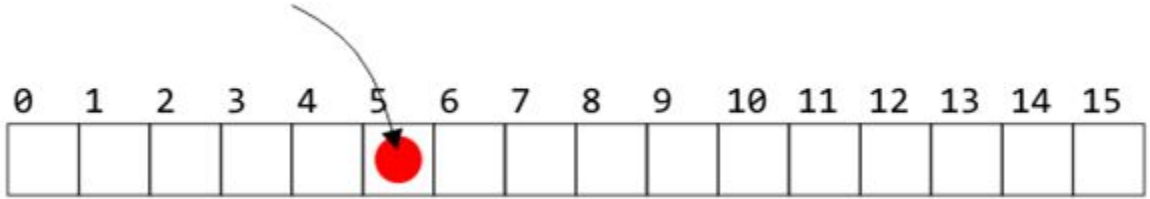
We will deal with collisions by storing collisions elsewhere

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

Open Addressing

Suppose an object hashes to bin 5

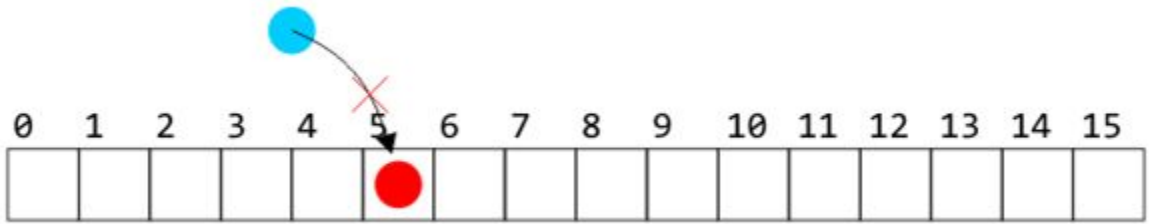
If bin 5 is empty, we can copy the object into that entry



Open Addressing

Suppose, however, another object hashes to bin 5

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

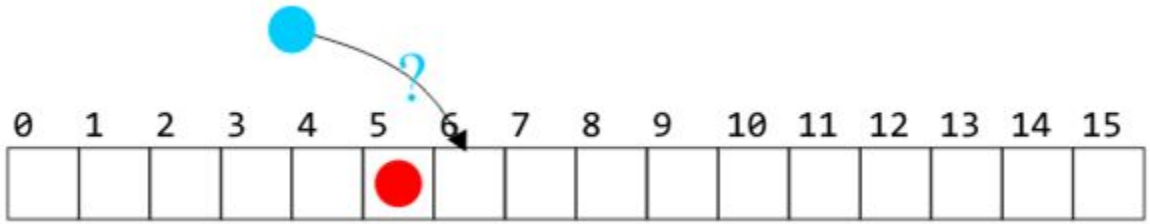


Open Addressing

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

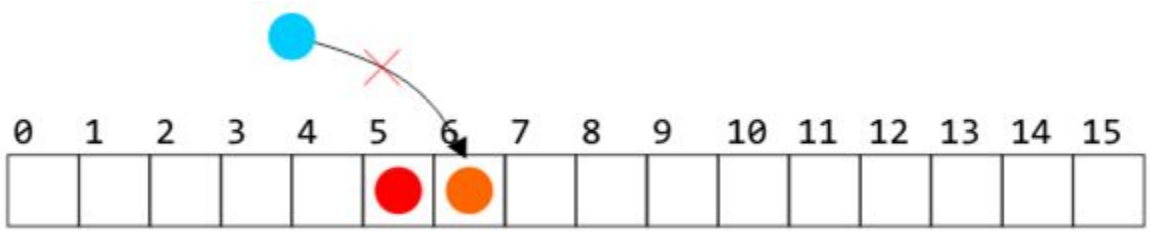


Open Addressing

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

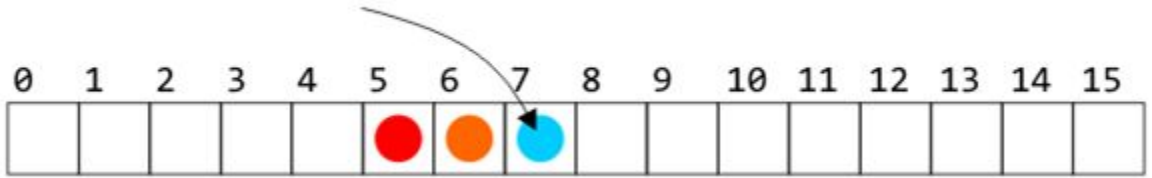
The rule must be simple to follow—i.e., fast



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



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
- There are numerous strategies for defining the order in which the bins should be searched:
  - Linear probing
  - Quadratic probing
  - Double hashing

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)

Insertion

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

Insert these numbers into this initially empty hash table:

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

Insertion

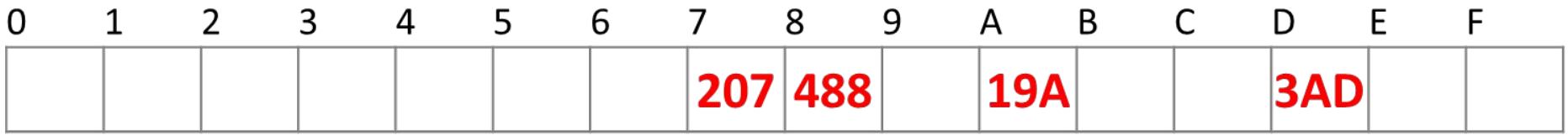
Start with the first four values:

19A, 207, 3AD, 488

Insertion

Start with the first four values:

19A, 207, 3AD, 488

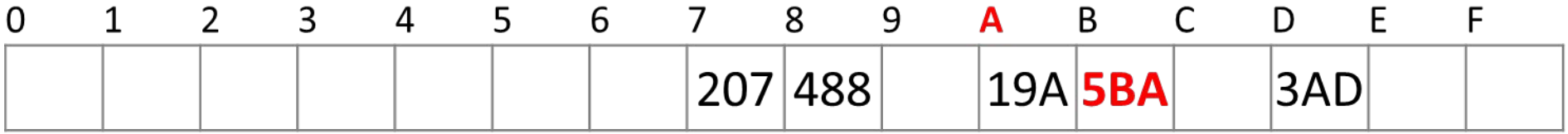


Insertion

Next we must insert 5BA

Bin A is occupied

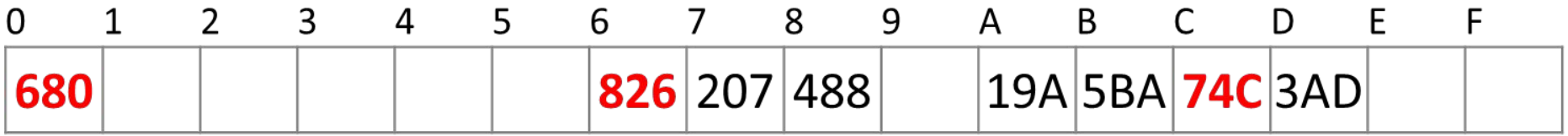
We search forward for the next empty bin



Insertion

Next we are adding 680, 74C, 826

All the bins are empty—simply insert them

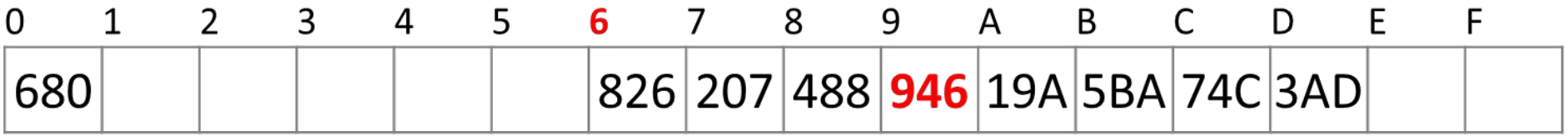


Insertion

Next, we must insert 946

Bin 6 is occupied

The next empty bin is 9

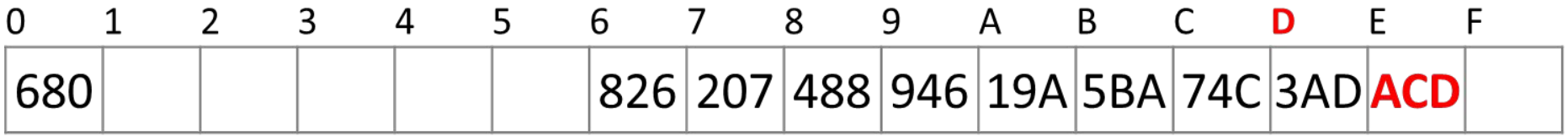


Insertion

Next, we must insert ACD

Bin D is occupied

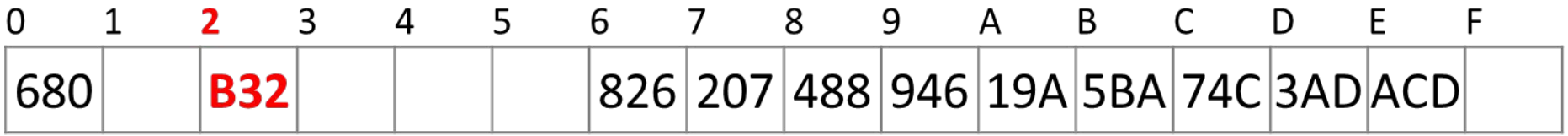
The next empty bin is E



Insertion

Next, we insert B32

Bin 2 is unoccupied



Insertion

Next, we insert C8B

Bin B is occupied

The next empty bin is F

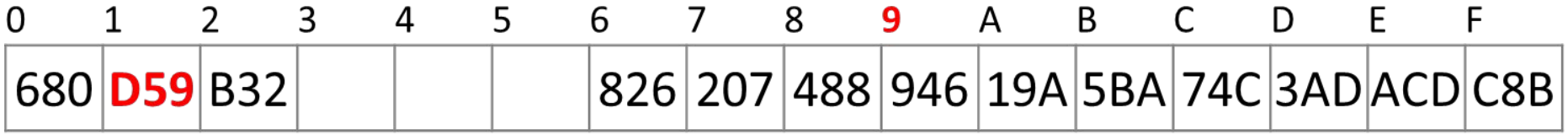
0	1	2	3	4	5	6	7	8	9	A	**B**	C	D	E	F
680		B32				826	207	488	946	19A	5BA	74C	3AD	ACD	**C8B**

Insertion

Next, we insert D59

Bin 9 is occupied

The next empty bin is 1



Insertion

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

Insertion

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

Searching

Testing for membership is similar to insertions:

- Start at the appropriate bin, and searching forward until
  - The item is found,
  - An empty bin is found, or
  - We have traversed the entire array

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

Quadratic Probing

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;
      // ...
  }
```

Quadratic Probing

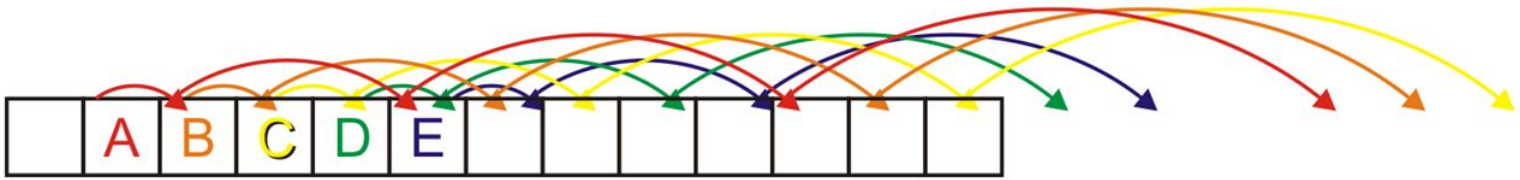
Quadratic probing suggests moving forward by different amounts

For example,

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

Problem:

Will initial + k*k step through all of the bins?



Generalization

More generally, we could consider an approach like:

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

Generalization

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

c1 = c2 = ½

```
int initial = hash_M( x.hash(), 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

Generalization

There is an even easier means of calculating this approach

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

Double Hashing

- Problems with linear and quadratic probing
  - Linear probing causes **primary clustering**
  - Quadratic probing causes **secondary clustering**
- An alternate solution?
  - Give each object (with high probability) a different jump size

Double Hashing

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;
  }
```

Double Hashing

The jump size and the number of bins M must be coprime

- They must share no common factors

There are two ways of ensuring this:

- Make M a prime number
- Restrict the prime factors

Double Hashing 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)

Insert these numbers into this initially empty hash table

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

Double Hashing Example

Start with the first four values:

19A, 207, 3AD, 488

Double Hashing 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		

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

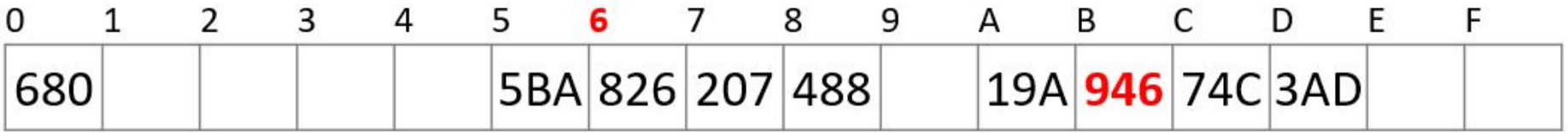
Double Hashing Example

Next, we must insert 946

Bin 6 is occupied

The second digit is 4, which is even

The jump size is 4 + 1 = 5



Double Hashing 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

Double Hashing 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		

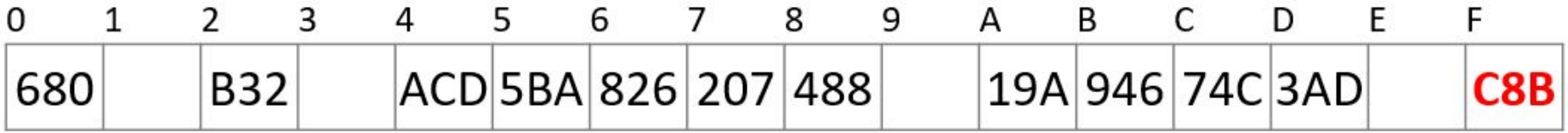
Double Hashing Example

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 $9 - 10_{16} = -7$



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

Double Hashing Example

Inserting D59, 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

Double Hashing Example

Finally, insert E9C

Bin C is occupied

The jump size is 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	D59	19A	946	74C	3AD	**E9C**	C8B

The sequence of bins is C, 5, E

Double Hashing 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

End of Lecture