Open addressing

Waqar Ahmad
Department of Computer Science
Syed Babar Ali School of Science and Engineering
Lahore University of Management Sciences (LUMS)

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

The hashing process

Object

Techniques vary...

32-bit integer

Modulus, mid-square, multiplicative, Fibonacci

Map to an index $0, ..., M - 1$

Deal with collisions

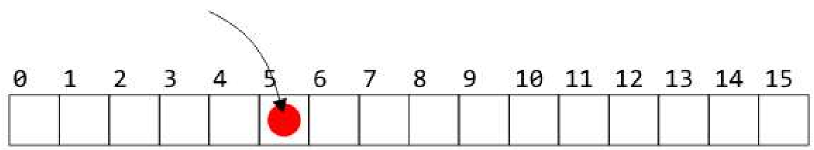
Chained hash tables
Open addressing

Linear probing
Quadratic probing
Double hashing

Open Addressing

Suppose an object hashes to bin 5

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

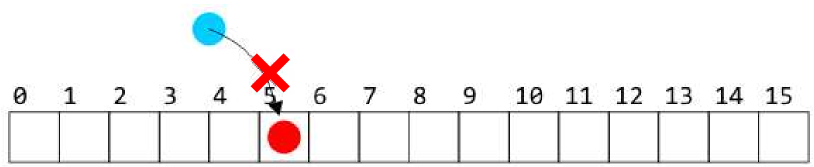


Open addressing is a collision resolution method used in hash tables where all elements are stored directly within the table itself, rather than using external linked lists as in chained hashing. When a collision occurs (i.e., two objects hash to the same index), open addressing finds another empty spot within the table to store the object, following a predetermined sequence. This keeps all the elements within a fixed-size array, making retrieval potentially faster but requiring a strategy to handle collisions effectively.

Open Addressing

Suppose, however, another object hashes to bin 5

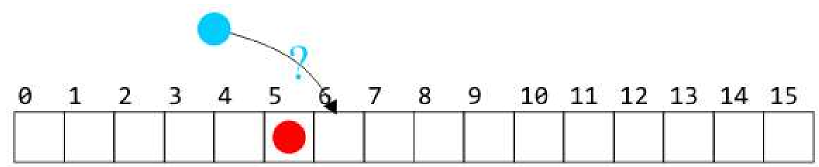
– Without a linked list, we cannot store the object in that bin



Open Addressing

We could have a rule which says:

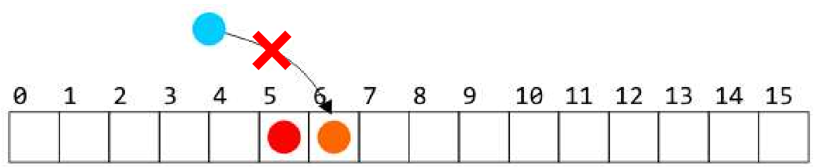
- Look in the next bin to see if it is occupied
- Such a rule is *implicit*—we do not follow an explicit link



Open Addressing

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

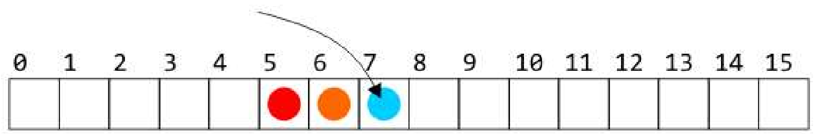
- For example, continue searching until the first empty bin is found
- The rule must be simple to follow—*i.e.*, fast



Open Addressing

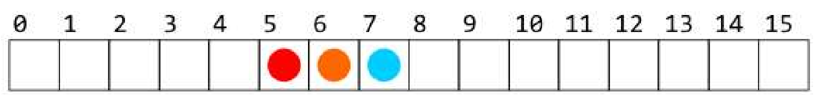
We could then store the object in the next location

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



Open Addressing

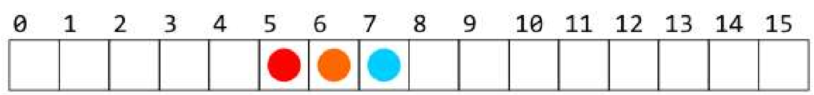
Whatever rule we use in placing an object, must also be used when searching for or removing objects



Open Addressing

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

– We should not, 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 alternate strategies, as well:

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

Linear Probing

Our first scheme for open addressing:

- Linear probing—keep looking ahead one cell at a time

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:

- Let’s say the least-significant digit is the primary hash function (bin)
- Example: for $6B72A_{16}$, the initial bin is **A**

Insertion

Insert these numbers into this initially empty hash table:

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

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

Example

Start with the first four values:

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

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

Example

Next we 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 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 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 insert 94**6**

- 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 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 insert AC**D**

- 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 B3**2**

– 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 $\lambda = 14/16 = 0.875$

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

Marking bins occupied

How can we mark a bin as empty?

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 may lead to 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 search 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 Search 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	**ACD**	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 C8B 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	**C8B**	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	**D59**

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: An Alternative Approach

- Use the concept of *lazy deletion*
  - Mark a bin as `ERASED`; however, when searching, treat the bin as occupied and continue
    - We should have a separate ternary-valued flag for each bin
      - Unoccupied
      - Occupied
      - Erased

We must also modify insert, as we may place new items into either
  - Unoccupied bins
  - Erased bins

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

Run-time analysis

- Our goal is to keep all operations $\Theta(1)$
- Unfortunately, as λ grows, so does the run time

- We have three choices:
  - Choose M (size of hash table) large enough so that we will not pass the load factor beyond a certain value
    - This could waste memory
  - Double the number of bins if the chosen load factor is reached
  - Choose a different strategy from linear probing
    - Two possibilities are:
      - quadratic probing and
      - double hashing

Quadratic Probing

Background

Linear probing:

– Look at bins k, $k + 1$, $k + 2$, $k + 3$, $k + 4$, …

Background

- Linear probing causes primary clustering:
  - With more insertions, the contiguous regions (or *clusters*) get larger
  - 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;
    // ...
}
```

Description

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;
}
```

Generalization

More generally, we could consider an approach like the following for quadratic probing:

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

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

Table Size (M) and Probe Function

- The right combination of probe function and table size will visit large number of slots in the hash table.
- If we make the table size *M* = *p,* where p is a prime number, quadratic probing (k^2) is guaranteed to iterate through $\left\lceil \frac{p}{2} \right\rceil$ entries.
  - If the table is less than half full, we can be certain that a free slot will be found
  - Doubling the number of bins is difficult
    - What is the next prime after 2 × 263?
- If we ensure *M* = 2^m (i.e., the table size is a power of 2) and the probe function is $\frac{k^2+k}{2}$, then every slot in the table will be visited by the probe function (See the next slide)

Example: Using $M = 2^m$

Here is an example of using a quadratic probing approach

```
int bin = hash_M( x.hash(), M );

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

– Recall that $\frac{k^2 + k}{2} = \sum_{j=0}^{k} j$, so just keep adding the next highest value

If M is a prime number: When M is prime, quadratic probing will iterate through half of the table without repeating, ensuring a free slot is found if the table is less than half full.
If M is a power of 2: The table size can also be a power of 2, but the probe function might need to use a different pattern (such as k*(k+1)/2) to ensure that all slots are visited without repeating the same sequence.

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 $6B7\mathbf{A}_{16}$, 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 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 3**2**

– 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	A	B	C	D	E	F
80		32		**7A**		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	**A**	B	C	D	E	F
80		32		7A	**BF**	26	07	88	46	9A	BA	4C	AD		C9

Example

Finally, we insert 9**C**

- 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 $\lambda = 14/16 = 0.875$

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

Erase

Can we erase an object?

- Consider erasing 9A from this table
- There are $M - 1$ possible locations where an object which could have replaced 9A 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	~~AD~~		C9

Find

When searching, it is necessary to skip over this bin

- For example, find AD: D, E
  - find D9: 9, A, C, F, 3
  - find BF: F, 0, 2, 5

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

Modified insertion

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

- Unoccupied bins
- Erased bins

Implementation

Storing three states can be achieved using an enumerated type:

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

Now we can declare and initialize arrays:

```
bin_state_t state[M];

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

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

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

Computers 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

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 (make it odd if not already) is the secondary hash function (jump size)
- Example: for 6B7**2A**$_{16}$, the initial bin is **A** and the jump size is (**2**+1) **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

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

Example

Start with the first four values:

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

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 5**B****A**

- Bin **A** is occupied
- The jump size **B** is already odd

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 5**B****A**

- Bin **A** is occupied
- The jump size is **B** is already odd

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

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

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 C**8B**

- Bin **B** is occupied
- The jump size is **8** which is even, so 8 + 1 = 9 is odd

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

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

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

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 $\lambda = 14/16 = 0.875$

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

Erase

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 a bin marked as Erased

– 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

Continue until the desired value is found, an empty location is reached, or the entire table has been searched.

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

References and Acknowledgements

- The content provided in the slides are borrowed from different sources including Goodrich's book on Data Structures and Algorithms in C++, Cormen's book on Introduction to Algorithms, Weiss's book , Data Structures and Algorithm Analysis in C++, 3rd Ed., Algorithms and Data Structures at University of Waterloo (https://ece.uwaterloo.ca/~dwharder/aads/Lecture_materials/), https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/BinaryTreeTraversal.html and https://research.cs.vt.edu/AVresearch/hashing/quadratic.php.
- The primary source of slides is https://ece.uwaterloo.ca/~dwharder/aads/Lecture_materials, courtesy of Douglas Wilhelm Harder.
- .