Open Addressing

Kyunghan Lee
Networked Computing Lab (NXC Lab)
Department of Electrical and Computer Engineering
Seoul National University
https://nxc.snu.ac.kr
kyunghanlee@snu.ac.kr

Outline

- To handle collisions, chained hash tables require special memory allocation
  - Can we create a hash table without additional memory allocation?

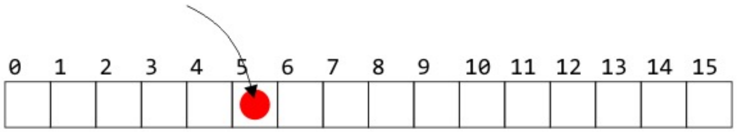
- We will deal with collisions by storing collisions in the same table
  - We will define a rule, dictating where to look next

Collision Handling in Hash Tables

- Common strategies to handle collisions in hash tables
  - Closed addressing: Store all elements with hash collisions in a secondary data structures (linked list, BST, etc.)
    - Chained hash table
  - Perfect Hashing: Choose a hash function to ensure that collisions don't happen (if possible)
  - Open addressing: Define a rule to locate the next cell
    - Linear probing, Quadratic probing, double hashing

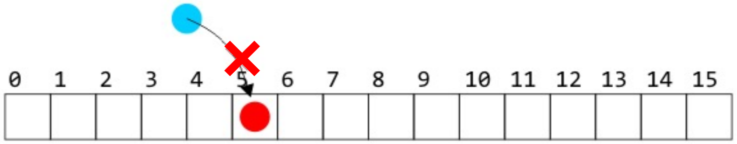
Open Addressing: Insert

- Suppose an object hashes to bin 5
  - If bin 5 is empty, we can store the object in bin 5



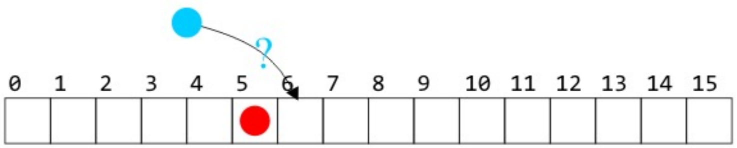
Open Addressing: Insert

- Suppose, however, another object hashes to bin 5
  - Without a linked list, we cannot store the object in bin 5



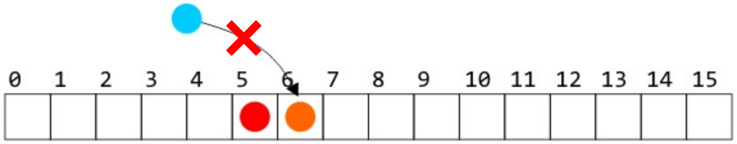
Open Addressing: Insert

- We could have a rule which says:
  - Look in the next bin to see if it is occupied



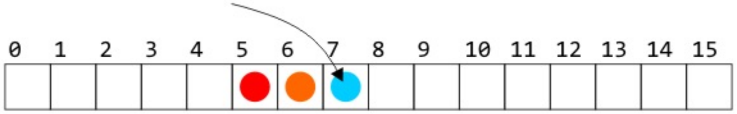
Open Addressing: Insert

- 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 — i.e., fast search



Open Addressing: Insert

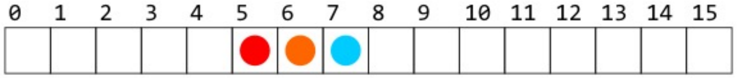
- We could then store the object in the next location
  - Problem: we can only store as many objects as there are entries in the array: the load factor $\lambda \leq 1$



Open Addressing: Supporting Other Operations

- The rule should support both search and remove.

- Recall that our goal is $\Theta(1)$ access times
  - Q. how do we avoid to access too many bins (on average)?



Open Addressing: Strategies

- 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

Linear Probing

Linear Probing

- The easiest method to probe 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, go back to the front (bin 0)

Linear Probing

- Consider a hash table with $M = 16$ bins

- Given a hexadecimal number as input:
  - Suppose the hash function outputs the least significant 4-bits of input
  - 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:

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 5BA
  - 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 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 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	E9C			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
  - Now the hash function outputs the least significant 5-bits of input
  - 680, B32, ACD, 5BA, 826, 207, 488, D59 may be immediately placed

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 λ = 14/32 = 0.4375
  - The average number of probes is 18/14 ≈ 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	E9C			826	207	488	946	19A	5BA	74C	3AD	ACD	C8B

Searching

- Searching for C8B

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

Searching

- Searching for C8B
  - 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	E9C			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	E9C			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	E9C	×		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	E9C			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	E9C			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, you should mark the cell "erased".
  - This "erased cell" is different from an empty cell---the search should not stop at an erased cell

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

- Each cell may be represented with the following states:
  - Occupied
  - Empty
  - Erased

- Your "cell" positioning algorithm should be different for "search" and "insert"

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≈16 % chance that an insertion will fill 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
680						826	207	488	946		C8B	74C	ACD	3AD			946	B32							D59	5BA	19A	E9C			

Primary Clustering

- There is a 5/32≈16 % chance that an insertion will fill Bin A
  - This causes two clusters to *coalesce* into one larger cluster of length 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
680						826	207	488	946	747	C8B	74C	ACD	3AD			946	B32							D59	5BA	19A	E9C			

Primary Clustering

- There is now a 11/32 ≈ 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

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			

- 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 $\frac{\ell+2}{M}$

Run-time analysis

- Recall: our goal is to keep all operations O(1).

- Which operations should we analyze?
  - Search
    - Unsuccessful search: After probing, we failed to find a key k in HT
    - Successful search: After probing, we found a key k in HT
  - Insert
    - The runtime would be the same as an unsuccessful search
  - Remove
    - The runtime would be the same as a successful search

Run-time Analysis: Unsuccessful Search

- **Theorem**

  Given a linear-probing hash table with the load factor λ, **the expected number of probes in an unsuccessful search is at most $1/(1-\lambda)$**, assuming uniform hashing

Run-time Analysis: Unsuccessful Search

- **Proof** (details in CLRS p274)
  - In an unsuccessful search
    - every probe (except the last) accesses an occupied cell, which does not contain the desired key
    - The last probe accesses an empty cell
  - Let the random variable X be the number of probes made in an unsuccessful search
  - Let the event A_i be the event that an i-th probe occurs, and it is to an occupied cell (which does not contain the desired key)

Thus, the event $\{X \geq i\}$ is

$A_1 \cap A_2 \cap \cdots \cap A_{i-1}$.

$$\Pr\{A_1 \cap \cdots \cap A_{i-1}\} = \Pr(A_1) \cdot \Pr(A_2 | A_1) \cdot \Pr(A_3 | A_1 \cap A_2) \cdot \cdots \cdot \Pr(A_{i-1} | A_1 \cap \cdots \cap A_{i-2})$$

$$\Pr(A_1) = \frac{n}{M}$$

$$\Pr(A_2 | A_1) = \frac{n-1}{M-1}$$

$$\Pr(A_j | A_1 \cap \cdots \cap A_{j-1}) = \frac{n-(j-1)}{M-(j-1)}$$

$$\Pr\{X \geq i\} = \frac{n}{M} \cdot \frac{n-1}{M-1} \cdots \frac{n-(j-2)}{M-(j-2)}$$

$$\leq \left(\frac{n}{M}\right)^{i-1} = \lambda^{i-1}$$

$$E(X) \leq \sum_{i=1}^{\infty} \Pr(X \geq i) \leq \sum_{i=1}^{\infty} \lambda^{i-1}$$

$$= 1 + \lambda + \lambda^2 + \cdots = \frac{1}{1-\lambda}$$

Run-time Analysis: Insertion

- **Corollary**

  Inserting an element into a linear-probing hash table with load factor λ requires at most $1/(1-\lambda)$ probes on average, assuming uniform hashing

- **Proof sketch**

  An unsuccessful search implies that an empty cell is found, which can be used for the insertion. So the insertion should take no more than the unsuccessful search.

Run-time Analysis: Successful Search

- **Theorem**
  - Given a linear-probing hash table, the expected number of probes in a successful search is at most

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

  - Assuming uniform hashing
  - Assuming that each key in the table is equally likely to be search for.

- **Proof sketch** (CLRS p276)
  - The successful search should take place after the insertion (w.r.t. key k)
  - The successful search would follow the same probing sequence as the insertion
  - So we take the average of the probing sequence in the insertion is the average number of successful probes

Run-time analysis

- The analysis shows that if we assume λ is constant, all operations are O(1) on average.

	Average	Worst
Search	O(1)	O(n)
Insert	O(1)	O(n)
Delete	O(1)	O(n)

- Still the analysis implicates that as λ gets bigger, the number of probes increases.
  - Q. What's the number of probes if the table is half full?
  - Q. what's the number of probes if the table is 90% full?

Run-time analysis

- The analysis implies:
  - Choose *M* large enough so that we will not pass the load factor
    - This could waste memory
  - Double the number of bins if the chosen load factor is reached
    - Rehashing will be required

- Q. Would other collision resolution methods help to reduce the number of probes?
  - It won't help the asymptotic complexity, but may help for some cases
  - We will cover quadratic probing next

Quadratic Probing

Primary Clustering in Linear Probing

- Recall Linear probing:
  - Look at bins k, $k + 1$, $k + 2$, $k + 3$, $k + 4$, …
  - Linear probing causes primary clustering
  - All entries follow the same search pattern for bins:

```
int initial = hash_M(x);
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);

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

Description

- Problem:
  - Will 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
```

Description

- Problem:
  - Will 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

Best in Theory: Making *M* Prime

- **Theorem:**

  If the table size is $M = p$ a prime number and a quadratic probing is used, the first $p/2$ probes are distinct.

- This theorem in fact implies that at least the half of slots will be visited before the probe sequence repeats.

Best in Theory: Making M Prime

- **Proof by contradiction:**

Suppose there is a slot, which is visited twice during the first $M/2$ probes.
Let i and j be such two visits, where $0 \le i < j \le \frac{M}{2}$.

$$(H + i^2)\% M = (H + j^2)\%M$$
$$(H + j^2) = (H + i^2) + kM$$
$$j^2 = i^2 + kM$$
$$j^2 - i^2 = kM$$
$$(j - i)(j + i) = kM$$

Because M is prime, either $(j - i)$ or $(j + i)$ should have a factor M.
In other words, either $(j - i)$ or $(j + i)$ should be divisible by M.

Case#1: $(j - i)$ is divisible by M.

From assumption, $i < j \le \frac{M}{2}$.

So $(j - i) < M$, which contradicts the case#1 constraint.

Case#2: $(j + i)$ is divisible by M.

From assumption, $i < j \le \frac{M}{2}$.

So $(j + i) < M$, which contradicts the case#2 constraint.

Best in Theory: Making *M* Prime

- **Engineering difficulties in using a prime M in practice:**
  - No optimized modulus operations
    - The modulus operator % is relatively slow
    - With a prime M, you cannot optimize with &, <<, or >>
  - Troublesome memory management
    - Memory Fragmentation
  - Doubling the number of bins is difficult:
    - You always need to find the next prime number
    - What is the next prime after 263?
      - ✓ You can't pick 2 * 263 as it's not a prime number

Generic Use

- More generally, we could consider an approach like:

```
int initial = hashM(x);

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

Practical Use: $M = 2^m$ with constraints

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

$$c_1 = c_2 = ½$$

```
int initial = hash_M(x);

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

- Note that k + k*k is always even
- This guarantees that all M entries are visited before the pattern repeats!
  - Proof sketch: Similar to the proof when M is prime

Practical Use: $M = 2^m$ with constraints

- 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
```

Practical Use: $M = 2^m$ with constraints

- There is an even easier means of calculating this approach

```
int bin = hash_M(x);

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

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	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 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

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 λ = 14/32 = 0.4375
  - The average number of probes is 20/14 ≈ 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

Run-time Analysis

- To summarize, quadratic probing shows the same asymptotic complexity as linear probing.

	Average	Worst
Search	O(1)	O(n)
Insert	O(1)	O(n)
Delete	O(1)	O(n)

Secondary clustering

- Advantage of quadratic probing over linear probing
  - Quadratic probing avoids primary clustering

One weakness with quadratic problem

- Objects initially placed in the same bin will follow the same sequence
- It forms yet another clustering, so called the secondary clustering
- Q. how would you solve this problem?

References

[1] Wikipedia, http://en.wikipedia.org/wiki/Hash_function
[2] Cormen, Leiserson, and Rivest, *Introduction to Algorithms*, McGraw Hill, 1990.
[3] Weiss, Data Structures and Algorithm Analysis in C++, 3rd Ed., Addison Wesley.

Reading Assignment #4 – Chapter 5 and 6

Quiz #3: 11/30 (4-5 questions, 50 mins, Lecture will follow)