ECE2810J

Data Structures and Algorithms

Hashing: Collision Resolution

Learning Objectives:

- Understand separate chaining
- Understand the general idea of open addressing
- Know three basic ways of open addressing and their advantages and disadvantages

Outline

- Collision Resolution: Separate Chaining

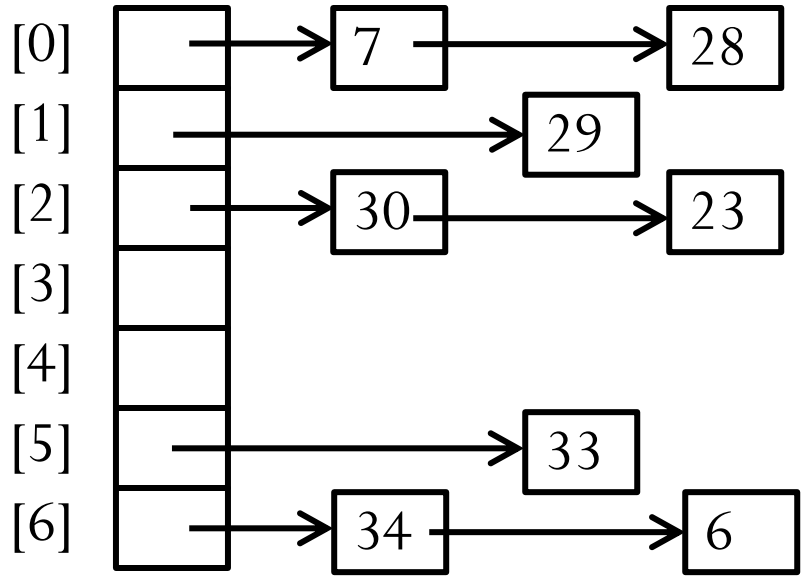
- Collision Resolution: Open Addressing
  - Linear Probing
  - Quadratic Probing and Double Hashing
  - Performance of Open Addressing

Collision Resolution Scheme

- **Collision-resolution scheme**: assigns distinct locations in the hash table to items involved in a collision.
- Two major schemes:
  - Separate chaining
  - Open addressing

Separate Chaining

- Each bucket keeps a **linked list** of all items whose home buckets are that bucket.
- Example: Put pairs whose keys are 6, 23, 34, 28, 29, 7, 33, 30 into a hash table with $n = 7$ buckets.
  - **`homeBucket = key % 7`**
  - **Note**: we insert object at the beginning of a linked list.



Separate Chaining

- **`Value find(Key key)`**
  - Compute **`k = h(key)`**
  - Search in the linked list located at the k-th bucket (e.g., check every entry) with the key.

- **`void insert(Key key, Value value)`**
  - Compute **`k = h(key)`**
  - Search in the linked list located at the k-th bucket. If found, update its value; otherwise, insert the pair at the beginning of the linked list in O(1) time.

Separate Chaining

- **`Value remove(Key key)`**
  - Compute **`k = h(key)`**
  - Search in the linked list located at the k-th bucket. If found, remove that pair.

Outline

- Collision Resolution: Separate Chaining

- Collision Resolution: Open Addressing
  - Linear Probing
  - Quadratic Probing and Double Hashing
  - Performance of Open Addressing

Open Addressing

- Reuse empty space in the hash table to hold colliding items.

- To do so, search the hash table in some systematic way for a bucket that is empty.
  - Idea: we use a sequence of hash functions $h_0, h_1, h_2, \ldots$ to probe the hash table until we find an empty slot.
    - I.e., we **probe** the hash table buckets mapped by $h_0(key)$, $h_1(key)$, …, in sequence, until we find an empty slot.
    - A widely-used way is to define $h_i(x) = h(x) + f(i)$

Open Addressing

- Three methods:
  - Linear probing:
    $h_i(x) = (h(x) + i) \% n$
  - Quadratic probing:
    $h_i(x) = (h(x) + i^2) \% n$
  - Double hashing:
    $h_i(x) = (h(x) + i*g(x)) \% n$

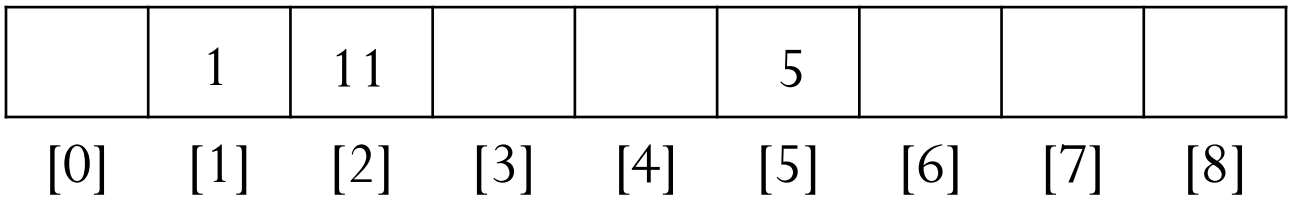
n is the hash table size

Linear Probing

$$h_i(key) = (h(key)+i) \% n$$

- Apply hash function h_0, h_1, …, in sequence until we find an empty slot.
  - This is equivalent to doing a linear search from **`h(key)`** until we find an empty slot.

- Example: Hash table size **`n = 9`**, **`h(key) = key%9`**
  - Thus **h_i`(key) = (key%9+i)%9`**
  - Suppose we insert 1, 5, 11, 2, 17, 21, 31 in sequence



How about 2?

Linear Probing

Example

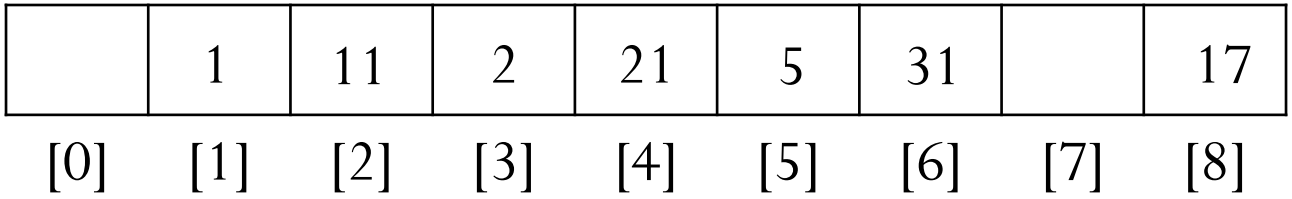
- Hash table size **`n = 9, h(key) = key%9`**
  - Thus **h_i`(key) = (key%9+i)%9`**
  - Suppose we insert 1, 5, 11, 2, 17, 21, 31 in sequence.

	1	11	2	21	5	31		17
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

  - **h_0`(2) = 2.`** Not empty!
  - So we try **h_1`(2) = 3.`** It is empty, so we insert there!
  - **h_0`(21) = 3.`** Not empty!
  - **h_1`(21) = 4.`** It is empty, so we insert there!
  - **h_0`(31) = 4.`** Not empty!
  - **h_1`(31) = 5.`** Not empty!
  - **h_2`(31) = 6.`** It is empty, so we insert there!

Linear Probing

find()



- With linear probing **`h_i(key) = (key%9+i)%9`**
  - How will you **search** an item with key = 31?
  - How will you **search** an item with key = 10?

- Procedure: probe in the buckets given by $h_0(key)$, $h_1(key)$, …, in sequence **until**
  - we find the key,
  - or we find an empty slot, which means the key is not found.

Linear Probing

remove()

	1	11	2	21	5	31		17
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

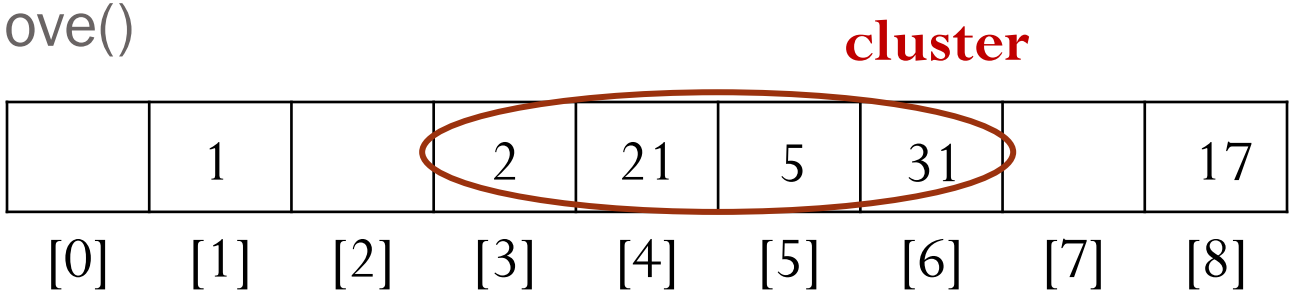
- With linear probing **h_i(key) = (key%9+i)%9**
  - How will you **remove** an item with key = 11?
  - If we just find 11 and delete it, will this work?

	1		2	21	5	31		17
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

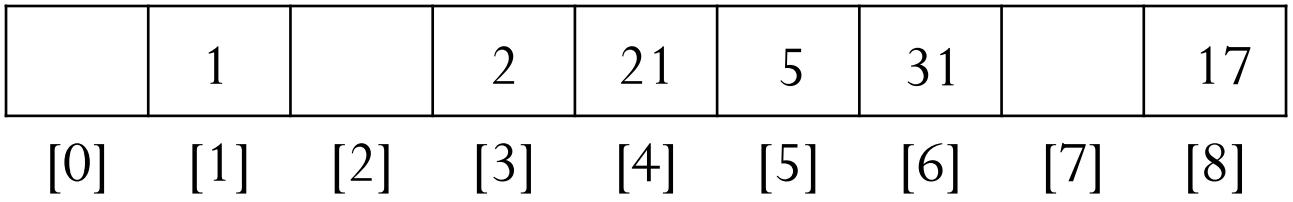
What is the result for searching key = 2 with the above hash table?

Linear Probing

remove()

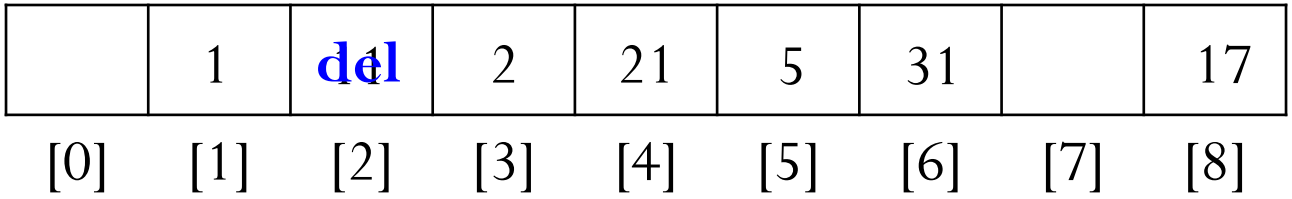


- After deleting 11, we need to **rehash** the following "cluster" to fill the vacated bucket.
- However, we cannot move an item **beyond** its **actual** hash position. In this example, 5 cannot be moved ahead.



Linear Probing

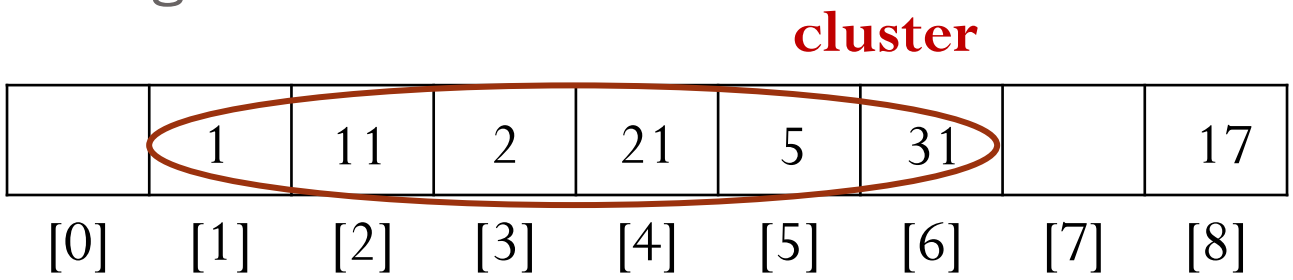
Alternative implementation of remove()



- **Lazy deletion**: we mark deleted entry as "**deleted**".
  - "deleted" is not the same as "empty".
  - Now each bucket has three states: "occupied", "empty", and "deleted".
- We can overwrite the "deleted" entry when inserting.
- When we **search**, we will keep looking if we encounter a "deleted" entry.

Linear Probing

Clustering Problem



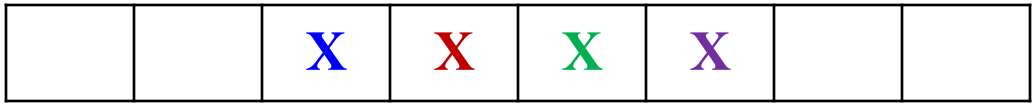
- Clustering: when **contiguous** buckets are all occupied.
- **Claim**: Any hash value inside the cluster adds to **the end** of that cluster.
- Problems with a **large** cluster:
  - It becomes more likely that the next hash value will collide with the cluster.
  - Collisions in the cluster get more expensive to resolve.

Linear Probing

Clustering Problem

- Assuming input size N, table size 2N:
  - What is the best-case cluster distribution?

  - What is the worst-case cluster distribution?



Which Statements Are Correct?

- Assuming input size N, table size 2N. Analyze the average number of probes to find an empty slot for best-case and worst-case clusters. Select all the correct answers.

A. The average number for best-case cluster is 1.5.

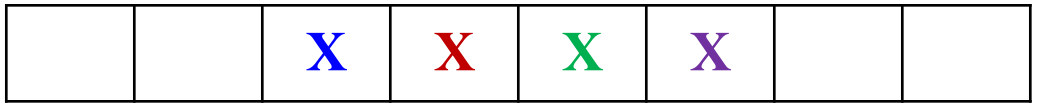
B. The average number for best-case cluster is 1.

C. The average number for worst-case cluster is roughly $\frac{1}{4}N$.

D. The average number for worst-case cluster is roughly $\frac{1}{2}N$.

Best Case

Worst Case



Outline

- Collision Resolution: Separate Chaining

- Collision Resolution: Open Addressing
  - Linear Probing
  - Quadratic Probing and Double Hashing
  - Performance of Open Addressing

Quadratic Probing

$$h_i(key) = (h(key)+i^2) \% n$$

- It is less likely to form large clusters.
- Example: Hash table size $n = 7, h(key) = key\%7$
  - Thus $h_i(key) = (key\%7+i^2)\%7$
  - Suppose we insert 9, 16, 11, 2 in sequence.

		9	16	11		2
[0]	[1]	[2]	[3]	[4]	[5]	[6]

  - $h_0(16) = 2$. Not empty!
  - $h_1(16) = 3$. It is empty, so we insert there.
  - $h_0(2) = 2$. Not empty!
  - $h_1(2) = 3$. Not empty!
  - $h_2(2) = 6$. It is empty, so we insert there.

Problem of Quadratic Probing

- However, sometimes we will never find an empty slot even if the table isn't full!
- Luckily, if the **load factor** $L \leq 0.5$, we are guaranteed to find an empty slot.
  - Definition: given a hash table with n buckets that stores m objects, its **load factor** is

$$L = \frac{m}{n} = \frac{\#\text{objects in hash table}}{\#\text{buckets in hash table}}$$

More on Load Factor of Hash Table

- Question: which collision resolution strategy is feasible for load factor larger than 1?
  - Answer: separate chaining.
  - Note: for open addressing, we require $L \leq 1$.
- Claim: $L = O(1)$ is a necessary condition for operations to run in constant time.

Double Hashing

$$h_i(x) = (h(x) + i*g(x)) \% n$$

- Uses 2 distinct hash functions.

- Increment **differently** depending on the key.
  - If h(x) = 13, g(x) = 17, the probe sequence is 13, 30, 47, 64, …
  - If h(x) = 19, g(x) = 7, the probe sequence is 19, 26, 33, 40, …
  - For linear and quadratic probing, the incremental probing patterns are **the same** for all the keys.

Double Hashing

Example

- Hash table size **`n = 7, h(key) = key%7,`**
  `g(key) = (5-key)%5`
  - Thus **h_i`(key) = (key%7+(5-key)%5*i)%7`**
  - Suppose we insert 9, 16, 11, 2 in sequence.

		9		11	2	16
[0]	[1]	[2]	[3]	[4]	[5]	[6]

  - **h_0`(16) = 2.`** Not empty!
  - **h_1`(16) = 6.`** It is empty, so we insert there.
  - **h_0`(2) = 2.`** Not empty!
  - **h_1`(2) = 5.`** It is empty, so we insert there.

Outline

- Collision Resolution: Separate Chaining

- Collision Resolution: Open Addressing
  - Linear Probing
  - Quadratic Probing and Double Hashing
  - Performance of Open Addressing

Performance of Open Addressing

- Hard to analyze rigorously.
- The runtime is dominated by the number of comparisons.
- The number of comparisons depends on the load factor L.
- Define the expected number of comparisons in an **unsuccessful search** as $U(L)$.
- Define the expected number of comparisons in a **successful search** as $S(L)$.

Expected Number of Comparisons

- Linear probing

$$U(L) = \frac{1}{2}\left[1 + \left(\frac{1}{1-L}\right)^2\right]$$

$$S(L) = \frac{1}{2}\left[1 + \frac{1}{1-L}\right]$$

L	$U(L)$	$S(L)$
0.5	2.5	1.5
0.75	8.5	2.5
0.9	50.5	5.5

$L \leq 0.75$ is recommended.

Expected Number of Comparisons

- Quadratic probing and double hashing

$$U(L) = \frac{1}{1 - L}$$

$$S(L) = \frac{1}{L} \ln \frac{1}{1 - L}$$

L	$U(L)$	$S(L)$
0.5	2	1.4
0.75	4	1.8
0.9	10	2.6

Which Strategy to Use?

- Both separate chaining and open addressing are used in real applications.

- Some basic guidelines:
  - If space is important, better to use open addressing.
  - If need removing items, better to use separate chaining.
    - **`remove()`** is tricky in open addressing.
  - In mission critical application, prototype both and compare.