# VE281

## Data Structures and Algorithms

Open Addressing; Universal Hashing

# Announcement

- Lecture Rescheduling
  - No course on Nov. 6, 8, 10, 13, 15.
  - Make-up courses on
    - Oct. 20, Oct. 27, Nov. 3, Nov. 24, Dec. 1 (Fridays): 2 pm – 3:40 pm.

# Outline

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

- Pathological Data Sets and Universal Hashing

# 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 $h_0$, $h_1$, $h_2$, . . . 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.
    - Generally, we could 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

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 11 | | | 5 | | | |
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] |

How about 2?

# Linear Probing

## Example

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

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

  - $\mathbf{h_0(2) = 2}$. Not empty!
  - So we try $\mathbf{h_1(2) = 3}$. It is empty, so we insert there!
  - $\mathbf{h_0(21) = 3}$. Not empty!
  - $\mathbf{h_1(21) = 4}$. It is empty, so we insert there!
  - $\mathbf{h_0(31) = 4}$. Not empty!
  - $\mathbf{h_1(31) = 5}$. Not empty!
  - $\mathbf{h_2(31) = 6}$. It is empty, so we insert there!

# Linear Probing

## find()

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

- With linear probing $\mathtt{h_i(key) = (key\%9+i)\%9}$
  - 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()

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

- 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?

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

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

# Linear Probing

## remove()

cluster

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

- 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.

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

# Linear Probing

## Alternative implementation of remove()

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

- **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

**cluster**

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

- 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?

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| X | | X | | X | | X | |

  - What is the worst-case cluster distribution?

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | X | X | X | X | | |

  - What's the average number of probes to find an empty slot for each case?

# Outline

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

- Pathological Data Sets and Universal Hashing

# 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.

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

  - $h_0(16) = 2$. 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 $\mathbf{h_i}$**`(key) = (key%7+(5-key)%5*i)%7`**
  - Suppose we insert 9, 16, 11, 2 in sequence.

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

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

# Outline

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

- Pathological Data Sets and Universal Hashing

# 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.

# Outline

- Collision Resolution: Open Addressing
  - Linear Probing
  - Quadratic Probing and Double Hashing
  - Performance of Open Addressing
- Pathological Data Sets and Universal Hashing

# Pathological Data Sets

- The **ideal** hash function spreads <u>**every**</u> data set out evenly.

- Does such **ideal** hash function exist?
  - No! For every hash function, there is a **pathological data set**.

- <u>Reason</u>: Fix a hash function $h: U \rightarrow \{0,1, \dots, n-1\}$
  - There exists a bucket $i$ such that at least $|U|/n$ elements of $U$ hash to $i$ under $h$…
  - … if data set drawn only from these, everything collides!

# Pathological Data Sets

- **<u>Given</u>**: A hash table with $n$ buckets stores $m$ items. Use separate chaining.
  - **<u>Question</u>**: If all $m$ items are mapped to the same bucket, what's the time complexity of **`find()`**?

- **<u>Answer</u>**: The hash table degrades to a linked list.
  - The time complexity for **`find()`** is $O(m)$.
  - This is actually the **worst-case** time complexity.

# Solution to Pathological Data Sets

- **Universal hashing**:
  - Design **a family** $H$ of hash functions such that for **all** data set $S$, "**almost all**" functions $h \in H$ spread $S$ out "**pretty evenly**".
  - Pick a hash function **randomly** from the family $H$.

# Universal Family of Hash Functions

- Definition: Let $H$ be **a set of hash functions** from $U$ to $\{0,1,2,\ldots,n-1\}$. $H$ is **universal** if and only if:
  - **For all** $x, y \in U$ with $x \neq y$,

$$\Pr_{h \in H}\big(h(x) = h(y)\big) \leq \frac{1}{n}$$

- In other words, **any** two keys of $U$ collide with probability at most $1/n$ when the hash function $h$ is chosen **uniformly at random** from $H$
- Note: keys $x$ and $y$ fixed. Random on hash function picked
  - At most 1/n of the total functions map $x$ and $y$ to the same bucket
- Collision probability is as small as our "gold standard" of **completely random hashing**

# Universal Family of Hash Functions

## Example

- Example #1: The set of **all** functions that map from $U$ to $\{0,1,2,\ldots,n-1\}$.
  - The family contain $n^{|U|}$ functions.
- Is this family universal?
- Yes! Because for any keys $x \neq y$, exactly $1/n$ of the total functions map $x$ and $y$ to the same bucket
  - Partition all the functions into $n^2$ subsets $S_{i,j}$ $(0 \leq i,j \leq n-1)$, where $S_{i,j}$ contains functions $h$ such that $h(x) = i$ and $h(y) = j$
  - The numbers of functions in all subsets are equal.

# Universal Family of Hash Functions

## Example

- Example #2: $\{h_0, h_1, \dots h_{n-1}\}$ where $h_i: U \rightarrow i$, i.e., for any $u \in U, h_i(u) = i$.
- Is this family universal?
- No! Because for any keys $x \neq y$, all functions map $x$ and $y$ to the same bucket, i.e.,

$$\Pr_{h \in H}\big(h(x) = h(y)\big) = 1$$

# Real Example: Hashing IP Addresses

- Let $U =$ IP address of the form $(x_1, x_2, x_3, x_4)$ with each $x_i \in \{0,1,\ldots,255\}$
- Let hash table size $n$ be a **prime** number and $n > 255$.
  - Could be close to a multiple of #objects in the hash table.

- Define one hash function $h_a$ per 4-tuple $a = (a_1, a_2, a_3, a_4)$ with each $a_i \in \{0,1,\ldots,n-1\}$.
  - $h_a(x_1, x_2, x_3, x_4) = (a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4)\ mod\ n$
  - There are $n^4$ such functions.

# A Universal Family of Hash Functions

- **Define** the family $H$ = all $n^4$ $h_a$'s, i.e.,

$$H = \{h_a | a_1, a_2, a_3, a_4 \in \{0,1, \dots, n-1\}\}$$

$$h_a(x_1, x_2, x_3, x_4) = (a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4) \bmod n$$

> **Theorem**: Family $H$ is universal.

# Proof

- Consider distinct IP addresses $(x_1, x_2, x_3, x_4)$ and $(y_1, y_2, y_3, y_4)$
- Assume $x_4 \neq y_4$. We need to show the collision probability for a randomly chosen function $h_a \in H$ is at most $1/n$, i.e.,

$$\Pr_{h_a \in H}(h_a(x_1, \dots, x_4) = h_a(y_1, \dots, y_4)) \leq \frac{1}{n}$$

- Note: collision happens when

$$a_1 x_1 + \cdots a_4 x_4 = a_1 y_1 + \cdots a_4 y_4 \ (mod\ n)$$

$$\Leftrightarrow a_4(x_4 - y_4) = \sum_{i=1}^{3} a_i (y_i - x_i) \ (mod\ n)$$

Next: let's fix random choice $a_1, a_2, a_3$, but $a_4$ still random

# Proof (cont.)

- Question: with $a_1, a_2, a_3$ fixed arbitrarily, how many choices of $a_4$ satisfy

$$a_4(x_4 - y_4) = \underbrace{\sum_{i=1}^{3} a_i(y_i - x_i)\ (mod\ n)}_{\text{fixed value}} ?$$

- Note:
  1. $0 \leq x_4 \neq y_4 \leq 255$
  2. $n > 255$ is prime
- Claim: For any $b \in \{0, \dots, n-1\}$, there is at most one $a_4 \in \{0, \dots, n-1\}$ that let $a_4(x_4 - y_4) = b\ (mod\ n)$
- Proof: for any $a_4{}' \in \{0, \dots, n-1\}$ and $a_4{}' \neq a_4$,

$$(a_4 - a_4{}')(x_4 - y_4) \neq 0 (mod\ n)$$

Imply $\Pr_{h_a \in H}(h_a(x_1, \dots, x_4) = h_a(y_1, \dots, y_4)) \leq \frac{1}{n}$

**Q.E.D.**