VE281

Data Structures and Algorithms

Hash Function; Collision Resolution

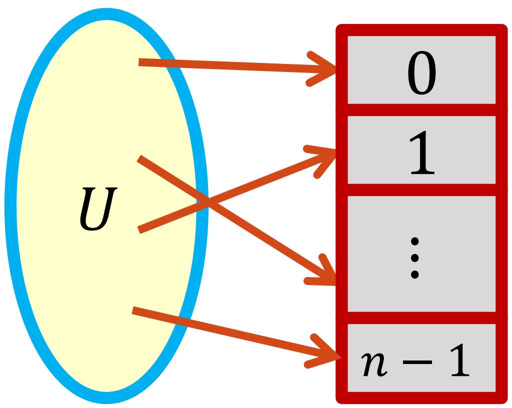
Outline

- Hash Function

- Collision Resolution
  - Separate Chaining
  - Open Addressing: Linear Probing
  - Open Addressing: Quadratic probing and double hashing

Review

- Hashing Basics
  - Hash table
  - Hash function $h: U \rightarrow \{0,1,\ldots,n-1\}$
  - Collision



Review: Hash Function

- Hash function design criteria
  - Must compute a bucket for every key in the universe.
  - Must compute the same bucket for the same key.
  - Should be easy and quick to compute.
  - Minimizes collision

- Hash function ($h(key)$) maps key to buckets in two steps:

1. Convert key into an integer in case the key is not an integer.
2. **Compression map**: Map an integer (hash code) into a home bucket.

Strings to Integers

- Simple scheme: adds up all the ASCII codes for all the chars in the string.
  - Example: t("He") = 72 + 101 = 173.

- Not good. Why?
  - Consider English words "post", "pots", "spot", "stop", "tops".

Strings to Integers

- A better strategy: Polynomial hash code taking **positional** info into account.

$$t(s[]) = s[0]a^{k-1} + s[1]a^{k-2} + \cdots + s[k-2]a + s[k-1]$$

where a is a constant.

- If $a = 33$, the hash codes for "post" and "stop" are

$$t(\text{post}) = 112 \cdot 33^3 + 111 \cdot 33^2 + 115 \cdot 33 + 116 = 4149734$$
$$t(\text{stop}) = 115 \cdot 33^3 + 116 \cdot 33^2 + 111 \cdot 33 + 112 = 4262854$$

Strings to Integers

$$t(s[]) = s[0]a^{k-1} + s[1]a^{k-2} + \cdots + s[k-2]a + s[k-1]$$

- Good choice of a for English words: 31, 33, 37, 39, 41
  - What does it mean for a to be a **good** choice? Why are these particular values **good**?
  - Answer: according to statistics on 50,000 English words, each of these constants will produce less than 7 collisions.
- In Java, its **`string`** class has a built-in **`hashCode()`** function. It takes $a = 31$. Why?
  - Multiplication by 31 can be replaced by a shift and a subtraction for **better performance**: **`31*i == (i << 5) - i`**

Hash function criteria: Should be easy and quick to compute.

Compression Map

- Map an integer (hash code) into a home bucket.
- The most common method is by **modulo arithmetic**.

```
homeBucket = c(hashcode) = hashcode % n
```

where n is the **number of buckets** in the hash table.

- Example: Pairs are (22,a), (33,b), (3,c), (55,d), (79,e). Hash table size is 7.

	(22,a)	(79,e)	(3,c)		(33,b)	(55,d)
[0]	[1]	[2]	[3]	[4]	[5]	[6]

Hashing by Modulo

- In practice, keys of an application tend to have a specific pattern
  - For example, memory address in computer is multiple of 4.

- The choice of the hash table size n will affect the distribution of home buckets.

Hashing by Modulo

- Suppose the keys of an application are more likely to be mapped into even integers.
  - E.g., memory address is always a multiple of 4.

- When the hash table size n is an **even** number, **even** integers are hashed into **even** home buckets.
  - E.g., n = 14: 20%14 = 6, 30%14 = 2, 8%14 = 8

- The bias in the keys results in a bias toward the **even** home buckets.
  - All **odd** buckets are **guaranteed** to be empty.
  - The distribution of home buckets is not uniform!

Hashing by Modulo

- However, when the hash table size n is **odd**, even (or odd) integers may be hashed into both odd and even home buckets.
  - E.g., n = 15: 20%15 = 5, 30%15 = 0, 8%15 = 8
    15%15 = 0, 3%15 = 3, 23%15 = 8
- The bias in the keys does not result in a bias toward either the odd or even home buckets.
  - Better chance of uniform distribution of home buckets.
- So **do not** use an even hash table size n.

Hashing by Modulo

- Similar **biased** distribution of home buckets happens in practice when the hash table size n is a multiple of small prime numbers.

- The effect of each prime divisor p of n **decreases** as p gets **larger**.

- Ideally, choose the hash table size n as a **large prime number**.

Outline

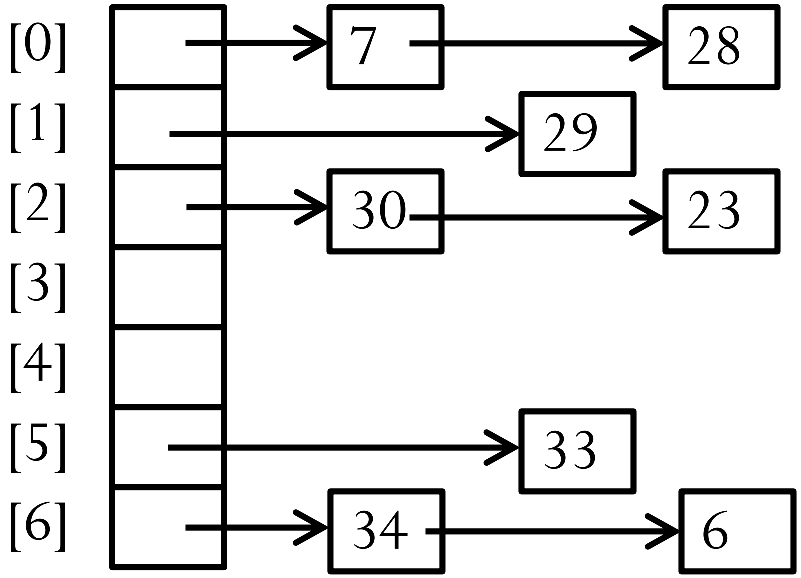
- Hash Function
- Collision Resolution
  - Separate Chaining
  - Open Addressing: Linear Probing
  - Open Addressing: Quadratic probing and double hashing

Collision Resolution Scheme

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

- Hash Function

- Collision Resolution
  - Separate Chaining
  - Open Addressing: Linear Probing
  - Open Addressing: Quadratic probing and double 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:
    $\mathbf{h_i(x) = (h(x) + i) \% n}$
  - Quadratic probing:
    $\mathbf{h_i(x) = (h(x) + i^2) \% n}$
  - Double hashing:
    $\mathbf{h_i(x) = (h(x) + i*g(x)) \% n}$

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 `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()

cluster

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

- 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.

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

Linear Probing

Alternative implementation of remove()

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

- **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

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

- 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 in both cases?

Outline

- Hash Function
- Collision Resolution
  - Separate Chaining
  - Open Addressing: Linear Probing
  - Open Addressing: Quadratic probing and double hashing

Quadratic Probing

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

- It is less likely to form large clusters.
- 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.

Aside: Load Factor of Hash Table

- Definition: given a hash table with n buckets that stores m objects, its **load factor** is

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

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

Quadratic Probing

Example

- Hash table size $\mathbf{n = 7}$, $\mathbf{h(key) = key\%7}$
  - Thus $\mathbf{h_i(key) = (key\%7+i^2)\%7}$
  - Suppose we insert 9, 16, 11, 2 in sequence.

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

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

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.

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

  - $\mathbf{h_0}$**`(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.

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.