VE281

Data Structures and Algorithms

Hashing

Review

- Dictionary: a collection of pairs **(key, element)**
  - Methods: **find()**, **insert()**, **remove()**
- Basics of Hashing
  - Hash table, hash function, collision
- Hash Function
  - Mapping non-integers into hash code
    - String to integers
  - Compression mapping by modulo arithmetic
    - How to choose the divisor M?

Outline

- Collision Resolution: Separate Chaining

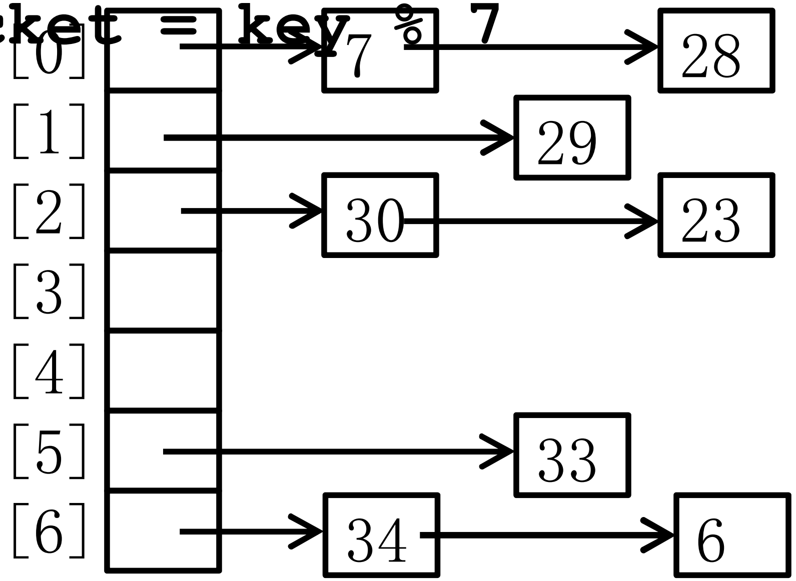
- Collision Resolution: Open Addressing
  - Linear probing
  - Quadratic probing and double hashing

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 M=7 buckets.
  - **homeBucket = key % 7**



Separate Chaining

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

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

Separate Chaining

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

Worst-Case Performance Analysis

- Suppose a hash table with M buckets stores N items.
  - What is the worst-case situation for find()?
  - What is the time complexity for find() in the worst case?

- Answer:
  - Worst case happens when all N items are mapped to the same bucket.
  - The time complexity for find() is $O(N)$.

Average-Case Performance Analysis

- Suppose a hash table with M buckets stores N items.
- Average list length for each bucket is N/M.
  - $L = N/M$ is called the **load factor**.
- Average runtime for search

$$O(h()) + O(1) + O(L)$$

Computing hash function	Fetch the k-th bucket	Search the linked list

- Average number of comparisons for an **unsuccessful** search: L.
- Average number of comparisons for a **successful** search:

$$\frac{1}{L}\sum_{i=1}^{L} i = \frac{L+1}{2}$$

Outline

- Collision Resolution: Separate Chaining
- Collision Resolution: Open Addressing
  - Linear probing
  - 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: if there is a collision, apply another hash function from a predetermined set of hash functions $\{h_0, h_1, h_2, \ldots\}$ in sequence until there' s no collision.

- Generally, we could define $h_i(x) = h(x) + f(i)$
  - We probe the hash table buckets mapped by $h_0(key)$, $h_1(key)$, …, in sequence, until we

Open Addressing

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

Linear Probing

$$h_i(\mathbf{key}) = (h(\mathbf{key})+i) \ \% \ M$$

- 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 **M = 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 **M = 9**, **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 $\mathbf{h_i(key) = (key\%9+i)\%9}$
  - 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 $\mathbf{h_i(key) = (key\%9+i)\%9}$
  - 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.
- Any hash value inside the cluster adds to the end of that cluster.
- Are there any 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 time to find an empty slot in both cases?

Outline

- Collision Resolution: Separate Chaining

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

Quadratic Probing

- `h_i(key) = (h(key)+i^2) % M`
- 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.

Quadratic Probing

Example

- Hash table size **M = 7**, **h(key) = key%7**
  - Thus $\mathbf{h_i(key) = (key\%7+}\color{red}{i^2}\mathbf{)\%7}$
  - 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.

Double Hashing

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

- Uses 2 distinct hash functions.
- Increments **differently** depending on the key.
  - For linear and quadratic probing, the incremental probing patterns are the same for all the keys.

Double Hashing

Example

- Hash table size **M = 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!
  - $\mathbf{h_2(2) = 5}$. It is empty, so we insert there.

Average Number of Comparisons

- It usually depends on the load factor $L = N/M$, where N is the number of items in the hash table and M is the size of the hash table.

- Define average number of comparisons in an **unsuccessful search** as $U(L)$.
- Define average number of comparisons in a **successful search** as $S(L)$.
- For separate chaining, we have

$$U(L) = L, \qquad S(L) = \frac{L+1}{2}$$

Average 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.

Average 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