CSCI 204: Data Structures & Algorithms

Revised by Xiannong Meng based on textbook author's notes

1

Hash Maps Introduction

Revised based on textbook author's notes.

Introduction

- When discussing search we saw:
  - linear search – O(n)
  - binary search – O(log n)
- Can we improve the search operation to achieve better than O(log n) time?

3

Comparison-Based Searches

- To locate an item, the target search key has to be compared against the other keys in the collection.
  - O(log n) is the best that can be achieved in comparison-based search.
  - We must use a different technique if we want to improve the search time.

4

Hashing

- The process of mapping a search key to a limited range of array indices.
  - The goal is to provide direct access to the keys.
  - **hash table** – the array containing the keys.
  - **hash function** – maps a key to an array index.

5

Hashing Example

- Suppose we have a list of popular fruits, we want to find if a particular type of fruit is in our inventory.
- Apple, Banana, Grape, Orange, Pear, Pineapple, Strawberry.
- We could use an array of 26 elements, each is index by the first letter of the fruit name, assuming no repetition. We can simply check for **fruit[name[0]]**!

6

Hashing Example

- Suppose we have the following set of keys

```
765, 431, 96, 142, 579, 226, 903, 388
```

a hash table, T, with M = 13 elements.

- We can define a simple hash function h()
- h(765) -> 11, h(431) -> 2, …

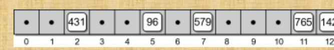
```
h(key) = key % M
```


Adding Keys

- To add a key to the hash table:
  - Apply the hash function to determine the array index in which the key should be stored.

```
h(765) => 11
h(431) => 2
h(96) => 5
h(142) => 12
h(579) => 7
```

  - Store the key in the given slot.

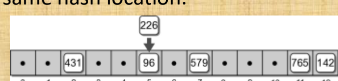


Collisions

- What happens when we attempt to add key 226?

```
h(226) => 5
```

- **collision** – when two or more keys map to the same hash location.

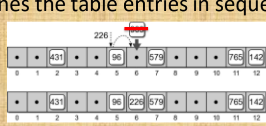


Resolving collisions

- There are in general two approaches to resolve collisions,
  - **Closed hashing**: find an open spot within the hash table to store the new element
  - **Open hashing**: create a structure, e.g., a list, or a tree, in the hashed spot to store the elements that have the same hashing key
- We first concentrate on closed hashing.

Closed hashing: probing

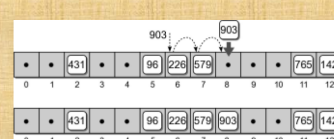
- If two keys map to the same table entry, we must resolve the collision to find another available slot.
  - **linear probe** – simplest approach which examines the table entries in sequential order.



Probing

- Consider adding key 903 to our hash table.

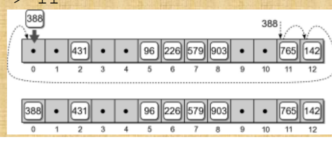
```
h(903) => 6
```



Probing

- If the end of the array is reached during the probe, it wraps around to the first entry and continues.
  - Consider adding key 388 to our hash table.

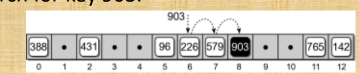
`h(388) => 11`



13

Searching

- Searching a hash table for a specific key is very similar to the add operation.
  - Target key is mapped to an initial slot.
  - See if the slot contains the target.
  - Otherwise, apply the same probe used to add keys to locate the target.
- Example: search for key 903.



14

Searching

- What if the key is not in the hash table?

- The probe continues until either:
  - a null reference is reached, or
  - all slots have been examined.

15

Deleting Keys

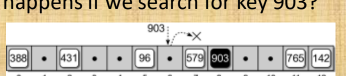
- Deleting a key from a hash table is a bit more complicated than adding keys.
  - We can search for the key to be deleted.
  - But we cannot simply remove it by setting the entry to `None`.

16

Incorrect Deletion

- Suppose we simply remove key 226 from slot 6.

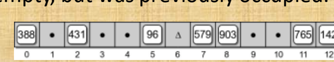
- What happens if we search for key 903?



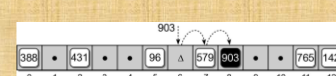
17

Correct Deletion

- We use a special flag to indicate the entry is now empty, but was previously occupied.



- When searching a hash table, the probe must continue past the slot(s) with the special flag.



18

Clustering

- The grouping of keys in a common area.
  - As more keys are added to the hash table, more collisions are likely to occur.
  - Clusters begin to form due to the probing required to find an empty slot.
  - As a cluster grows larger, more collisions will occur.
- **primary clustering** – clustering around the original hash position.

19

Probe Sequence

- The order in which the hash entries are visited during a probe.
  - The linear probe steps through the entries in sequential order.
  - The next array slot can be represented as

```
slot = (home + i) % M
```

  - where
    - `i` is the i[th] probe.
    - home is the **home position** of the original key

20

Modified Linear Probe

- We can improve the linear probe by changing the step size to some fixed constant.

```
slot = (home + i * c) % M
```

- Suppose we set c = 3 to build the hash table.

```
h(765) => 11        h(579) => 7
h(431) => 2         h(226) => 5   => 8
h(96)  => 5         h(903) => 6
h(142) => 12        h(388) => 11  => 1
```

•	388	431	•	•	96	903	579	226	•	•	765	142
0	1	2	3	4	5	6	7	8	9	10	11	12

21

Quadratic Probing

- A better approach for reducing primary clustering.

```
slot = (home + i**2) % M
```

- Increases the distance between each probe in the sequence.
- Example:

```
h(765) => 11     h(579) => 7
h(431) => 2      h(226) => 5   => 6
h(96)  => 5      h(903) => 6   => 7  => 10
h(142) => 12     h(388) => 11  => 12 => 2  => 7 => 1
```

•	388	431	•	•	96	226	579	•	•	903	765	142
0	1	2	3	4	5	6	7	8	9	10	11	12

22

Computations from last slide

- Quadratic probing

```
h(765) => 11     h(579) => 7
h(431) => 2      h(226) => 5   => 6
h(96)  => 5      h(903) => 6   => 7  => 10
h(142) => 12     h(388) => 11  => 12 => 2  => 7 => 1
```

```
h(226) => 5, second (5 + 1^2) % M => 6
h(903) => 6, second (6 + 1^2) % M => 7, third (6 + 2^2) % M => 10
h(388) => 11, second (11 + 1^2) % M => 12,
     third (11 + 2^2) % M => 2, fourth(11 + 3^2) % M => 7,
     fifth (11 + 4^2) % M => 1
```

•	388	431	•	•	96	226	579	•	•	903	765	142
0	1	2	3	4	5	6	7	8	9	10	11	12

Quadratic Probing

- Reduces the number of collisions.
- Introduces the problem of **secondary clustering**.
  - When two keys map to the same entry and have the same probe sequence.
- Example: add key 648
  - hashes to entry 11
  - follows the same sequence as key 388

•	388	431	•	•	96	226	579	•	•	903	765	142
0	1	2	3	4	5	6	7	8	9	10	11	12

24

Double Hashing

- When a collision occurs, a second hash function is used to build a probe sequence.

```
slot = (home + i * hp(key)) % M
```

  - Step size remains a constant throughout the probe.
  - Multiple keys that have the same home position, will have different probe sequences.

25

Double Hashing

- A simple choice for the second hash function.

```
hp(key) = 1 + key % P
```

- Example: let P = 8

```
h(765)  => 11         h(579)  => 7
h(431)  => 2          h(226)  => 5    => 8
h(96)   => 5          h(903)  => 6
h(142)  => 12         h(388)  => 11   => 3
```

0	1	2	3	4	5	6	7	8	9	10	11	12
•	•	431	388	•	96	903	579	226	•	•	765	142

26

Computations from last slide

- Double hashing
  - **slot = (home + i * hp(key)) % M, e.g., M==13**
  - **hp(key) = 1 + key % P, e.g., P == 8**

```
h(765)  => 11         h(579)  => 7
h(431)  => 2          h(226)  => 5    => 8
h(96)   => 5          h(903)  => 6
h(142)  => 12         h(388)  => 11   => 3
```

```
h(226) => 5, double hashing [(5+1*(1+226))%P] % M => 8
h(388) => 11, double hashing [(11+1*(1+388)%P] % M  => 3
```

0	1	2	3	4	5	6	7	8	9	10	11	12
•	•	431	388	•	96	903	579	226	•	•	765	142