CS350: Data Structures

Hash Tables

James Moscola
Department of Physical Sciences
York College of Pennsylvania

Hash Tables

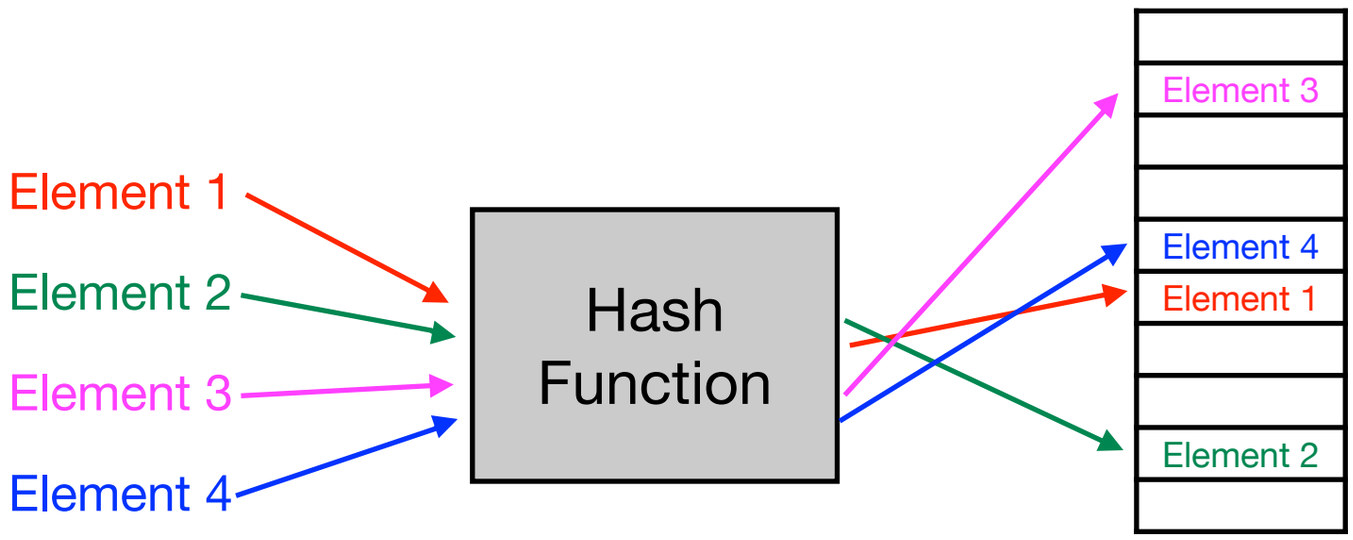
- **Although the various tree structures are fast, sometimes they are not fast enough**

- **Want a data structures that can perform `search`, `insert`, and `remove` in O(1) time on average**
  - Willing to sacrifice some of the functionality offered by trees

- **Keys are not kept in any order, therefore, some operations may take a long time**
  - Find minimum/maximum values in a data set takes O(N)

Hash Tables

- **Hash tables are typically represented as arrays of fixed size**
- **Each element that is to be inserted into a hash table has some signature, or hash code, associated with it**
- **The hash code is used as an index to insert the element into a table**
  - Example: Creating a hash code for a string -
    - ASCII characters can be represented as numbers
      - “Hello” --> 72 101 108 108 111
    - Convert sequence of numbers into a hash code N using some hash function
      - $N = 72 \cdot c^4 + 101 \cdot c^3 + 108 \cdot c^2 + 108 \cdot c^1 + 111 \cdot c^0$



Hash Function

- **Computation of the hash code is performed by a hash function**
  - Hash function is user defined
  - Selection of hash function is critically important to the performance of the hash table
    - Want a hash functions that is fast
      - Too complex of a hash function takes too long to compute
    - Want the hash function to create hash codes that are uniformly distributed throughout the hash table
      - Too simple of a hash function and many elements may hash to the same location of the hash table
- **A perfect hash function is a hash function that maps each element inserted into the hash table to a unique location**

Hash Function

- **Hash function maps keys of a given type to integers in a fixed interval `[0,N-1]`, where `N` is the size of the array representing the hash table**

- **A hash function is usually specified as the composition of two functions:**

  - Hash code map - takes key values and converts them to integers

    h_1: *keys* → *integers*

  - Compression map - takes the result from the hash code map and outputs a value in the range of `[0,N-1]`

    h_2: *integers* → `[0,N-1]`

- **To hash a key value x**

  $h(x) = h_2(\ h_1(x)\ )$

Compression map may be as simple as a modulus (e.g. $h_1(x) \bmod N$)

Idealized Hash Table

- **An idealized hash table ...**
  - would be an array exactly large enough to contain the data that is to be inserted (no wasted space)
  - would utilize a perfect hash function such that all entries into the table hash to a unique location of the table

- **To insert an element -**

```
i = hash(element)
hash_table_array[i] = element
```

- **To find an element -**

```
i = hash(element)
return hash_table_array[i]
```

- **To remove an element -**

```
i = hash(element)
hash_table_array[i] = null
```

In most cases, impossible to create an idealized hash table

Real Hash Table

- **In most cases, an idealized hash table is not possible**
  - Creating a perfect hash function is very difficult when size of table is just big enough to fit data

- **A real hash table is larger than will be needed to store the required data**
  - Simplifies the specification of the hash function since elements inserted into the table now have much more space to fill
    - Much of the space may go unused

- **Possible to have multiple elements that hash to the same location in the table**
  - This is known as a collision
  - To reduce the number of collisions, the size of the hash table array should be larger than actually required to store the expected number of elements inserted

Collision Handling

- **When more than one element inserted into a hash tables hashes to the same location in the table, a collision has occurred**

- **Collisions can be handled in several different ways**
  - Linear probing - handles collisions by placing the colliding elements in the next available table location (treat hash table array circularly)
    - Search for open position in hash table in the following sequence
      $h(x)$, $h(x)+1$, $h(x)+2$, $h(x)+3$, $h(x)+4$, ... , $h(x)+\mathrm{i}$
  - Quadratic probing - handles collisions similarly to linear probing, but searches for available table location using a quadratic sequence
    - Search for open position in hash table in the following sequence
      $h(x)$, $h(x)+1^2$, $h(x)+2^2$, $h(x)+3^2$, $h(x)+4^2$, ... , $h(x)+i^2$
  - Chaining - handles collisions by chaining hash table entries into a linked list

Linear Probing

- **Handles collisions by placing the colliding elements in the next available cell (treat hash table array circularly)**

$h(x)$, $h(x)+1$, $h(x)+2$, $h(x)+3$, $h(x)+4$, ... , $h(x)+i$

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

Collisions occur when inserting the key values 49, 58, 9

	After insert 89	After insert 18	After insert 49	After insert 58	After insert 9
0			49	49	49
1				58	58
2					9
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

When a collision occurs, scan the hash table for the next open address using a linear approach

Linear Probing

- **To `find` an element in the hash table**
  - Hash the search key to get an address into the hash table
  - If the key value in computed address is not the desired key, then scan the hash table from that point forward until the desired key is found (wrap around at the end of the table)
    - If an open address is found before the key is found, then the key does not exist in the hash table
    - Cannot delete items from table or some elements will no longer be searchable

- **To `delete` an element from the hash table**
  - Simply mark an item as deleted (lazy deletion)

Primary Clustering

- **Because of the way elements are inserted into the hash table when using linear probing, table may exhibit a type of clustering called primary clustering**
  - Large blocks (clusters) of occupied hash table locations are formed
  - Any key that hashes into the region occupied by the cluster will cause the cluster to enlarge
  - Can affect performance of hash table

Quadratic Probing

- **Handles collisions similarly to linear probing, but searches for available table location using a quadratic sequence**

$$h(x)\ ,\ h(x)+1^2\ ,\ h(x)+2^2\ ,\ h(x)+3^2\ ,\ h(x)+4^2\ ,\ \ldots\ ,\ h(x)+i^2$$

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

Collisions occur when inserting the key values 49, 58, 9

	After insert 89	After insert 18	After insert 49	After insert 58	After insert 9
0			49	49	49
1					
2				58	58
3					9
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

When a collision occurs, scan the hash table for the next open address using a quadratic approach

Quadratic Probing

- **The `find` and `delete` operations follow the same approach as with linear probing**
  - Using quadratic equation as opposed to linear equation
  - Continues to use lazy deletion

Performance of Quadratic Probing

- **Recall that a fast hash function is desired**
- **Quadratic probing suggests use of multiplication to probe for open location in hash table**
  - Multiplication is slow compared to addition/subtraction

$$h(x)\ ,\ h(x)+1^2\ ,\ h(x)+2^2\ ,\ h(x)+3^2\ ,\ h(x)+4^2\ ,\ \ldots\ ,\ h(x)+i^2$$

- **Multiplication can be removed from the sequence for quadratic probing**
  - If the location H_{i-1} is known, the location for H_i can be computed as:

$$H_i = H_{i-1} + 2i - 1 \ (mod\ M)$$

  - The multiplication by 2 can be computed with a simple shift
  - The mod operation can be replaced with a conditional subtraction
    - If $H_{i-1} + 2i - 1$ is < the size of the table, do nothing
    - If $H_{i-1} + 2i - 1$ is >= the size of the table subtract size of table

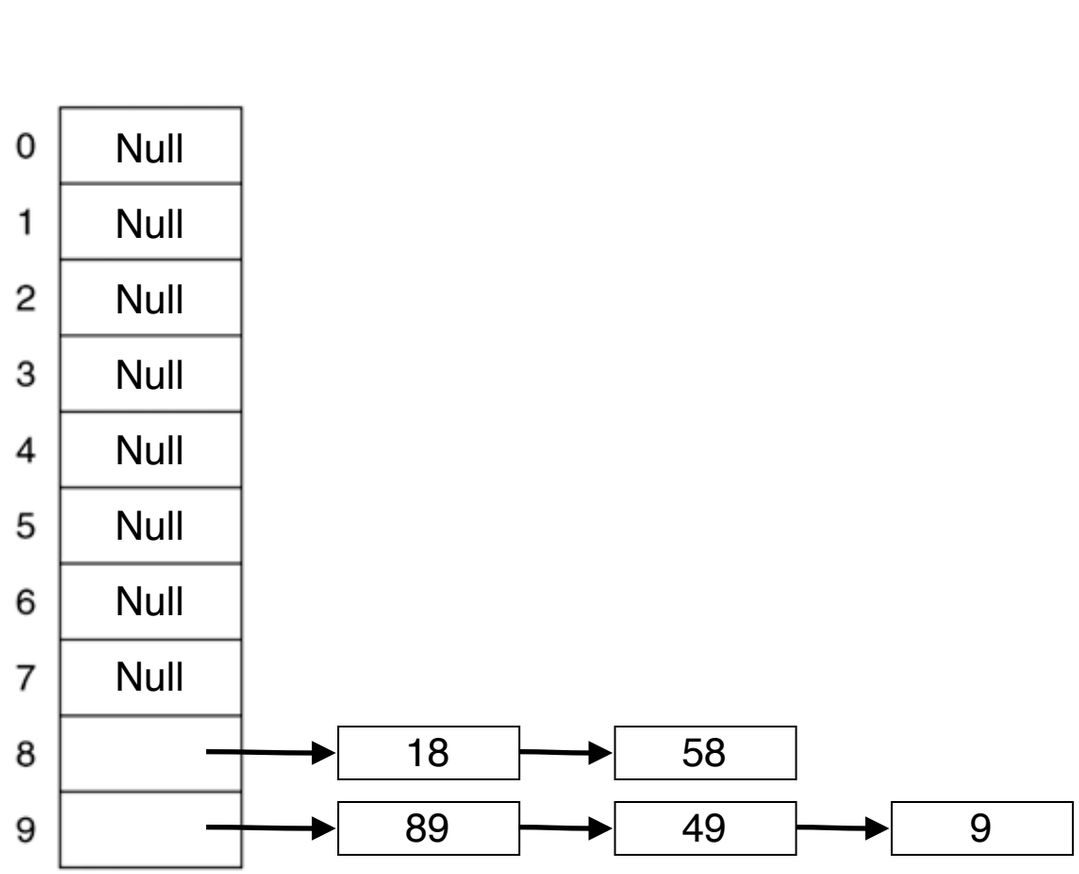
Secondary Clustering

- **Quadratic probing does eliminate primary clustering, however, does have secondary clustering**
  - Elements that hash to the same location will probe the same sequence of alternative locations

Separate Chaining

- **Handles collisions by chaining hash table entries into a linked list**
  - Each hash table location contains a pointer to a linked list
  - Elements that hash to the same location are stored in the linked list

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```



Separate Chaining

- **To `find` an element in the hash table**
  - Hash the search key to get an address into the hash table
  - Traverse the linked list at the computed address for the desired key
    - If the key is found, then return
    - If no linked list exists at the computed address, or if the search reaches the end of the linked list before finding the desired key, then the key does not exist in the hash table

- **To `delete` an element from the hash table**
  - Find the element to be deleted and remove it from the linked list

Determining Hash Table Size

- **How large should you make your hash table?**
  - Load factor of a hash table is `(#items_in_table/tableSize)`
    - Higher load factor results in greater number of collisions
    - Want to keep load factor < 0.5

- **Selecting the size of the table**
  - Table size should be a prime number approximately 2x the size of the number of elements