HashMaps

1332 Recitation: Week of June 2nd

Announcements

- First exam!
  - Scheduled for Thursday, June 4th
  - Will be open from 12AM EDT until 11:59 PM EDT
  - Timed (60 minutes)
  - More details+topic list in the announcement and the syllabus
- HW4 is available now!
  - Over HashMaps and will be due June 8th at 11:55 PM

Map ADT

- Designed to store key-value pairs (e.g. phone numbers and names, names and majors)
  - Keys are **immutable**
- Almost always implemented as a HashMap, which is backed by an array
  - You can insert, access, and remove key-value pairs in O(1) on average
  - There are other implementations, but we won't cover them in this class

Introduction

- Fun fact!
  - HashMap is the answer to 50% of interview questions
- Implementation of the Map ADT that is backed by an array
- There are similar classes like Hashtable and HashSet, but they are not in the scope of this class
- Computes a hash value for each key in a key-value pair and not the whole pair/ MapEntry object
- Hash: a “fingerprint” to identify the key (they are not 100% unique though)
- Keys are unique, but duplicate values are permitted
  - If (key1, value1) is inserted, followed by (key1, value2), (key1, value2) overwrites (key1, value1)

Computing the Hash

- Hashing function:
  - Applies some mathematical function onto an object to yield an int
    - E.g. a hash function for a string could be ASCII sum of all the characters
  - Objects that are equal should have equal hashes, but objects with equal hashes may not be equal
  - Should ideally be efficient to compute
  - In Java, you can use the **hashCode()** method to get the hash of an object
    - Every Object in Java has a hashCode() method
- Compression function:
  - Compress the hash of a key so that the key-value pair can fit within the bounds of the backing structure
  - index=Math.abs(key.hashCode()%arr.length)
    - **Mod the hash before taking abs** as that would fail when the hash is Integer.MIN_VAL

Adding

- Steps
  - Take the hash of the key
  - Compress the hash of the key so that it fits within the bounds of the array
    - Eg index = abs(hash(key) % table.length)
  - Add the key-value pair at this calculated index in the array
  - If the index is already occupied, we must resolve the collision
  - Enter collision resolution strategies

External Chaining

- Sometimes called “closed addressing” since items are stored at the index computed
- Each entry in the backing array is a LinkedList of key-value pairs
- Adding a key-value pair:
  - Compute the index
  - Iterate through the linkedList at the index to see if key already exists
  - We need to resize when there are a lot of elements in the HashMap to preserve quick access time

Load factor

- To preserve quick access time:
  - **Load factor** = size/capacity
    - If the load factor is too large, we resize before adding
  - We do not wait until the backing array is full because we start to lose quick access time as the HashMap gets mostly full

Resizing

- You must reinsert each element! This means retake the hash and recompress it to fit in the new backing array!
- Note that you don’t need to check for duplicates when reinserting because they were already checked when first added

External Chaining

- Remove
  - Compute the index
  - Iterate through the LinkedList to find node with that key
  - Remove node with that key from the LinkedList
- When adding, it doesn't matter if we add the the front or to the back (we need to iterate to check for duplicate keys anyway)
  - However, when resizing, we don't need to check for duplicates, so we can just add to the front
- Example!

Linear Probing

- Also called “open addressing” since items may not be stored at the index computed
- Backing array is an array of MapEntry objects
- High level add: compute index and if it’s occupied, try index + 1. Index + 2, and so on until something is null (wrap around if needed)
- High level remove: compute index, iterate until the key to remove is found, and mark it with a **DEL(deleted) flag** -- NO NULL
    - DEL flag is sometimes also known as “tombstone”

Linear Probing

- Adding
  - Compute the index
  - If there's an entry at index with different key, continue iterating to index + 1
  - If there's an entry at index that's deleted, then save the index and continue iterating
  - If there's an entry at index with the same key, replace the value
  - If the index is null, put the key-value pair there or the saved index if a deleted entry was found (which would be the first deleted entry)
  - Index = (orig_index + h) % arr.length, h = 0,1,2
- Removing a key:
  - Compute the index
  - Iterate from index until the entry with the key is found
  - Set the DEL flag to true
  - If the index is null, then that means that the key you are looking for is not in the HashMap
- When resizing, no need to copy deleted entries over
- Example

Quadratic Probing

- Similar to linear probing
- We iterate to index + 1, index + 4, index + 9
  - Index = (orig_index + h * h) % arr.length, h = 0,1,2
- Spreads out data in array better
- There are cases where adding to hash map with quadratic probing might loop infinitely
  - Adding 44 to [null, null, 9, 17, 32, null, 6]

Double hashing

- Use a second hash function that offsets the result of the initial hash function
- Index = (hash_1 + hash_2 * h) % arr.length, h = 0,1,2, ...
- Second hash must never equal 0
- Hash_2 doesn't need to be another hashCode() method; it just needs to be an independent hash
- Spreads out data in array better
- Can also infinitely loop
    - Resize after n iterations

Efficiencies

	Best/Average	Worst
Adding	O(1)	O(n)
Removing	O(1))	O(n)
Searching	O(1)	O(n)