CS 310: Hashing (Part I)

Connor Baker

February 26, 2019

| Implementation	get/set	add/del at end	add/del start	add/del mid	search	can grow?
Static Array	1	1	N	N	N	no
Dynamic Array	1	$\mathbf{1}^*$	N	N	N	no
Singly-Linked	N	1, N	1	N	N	yes
Doubly-Linked	N	1	1	N	N	yes
Stack	1 (top)	1 (pop)	1 (push)	-	-	yes
Queue	1 (getFront)	1 (enqueue)	1 (dequeue)	-	-	yes

*Amortized analysis

Motivation

- Task: suggest a data structure that can support retrieval and deletion of any value in constant time

    - `.add(T t)`, `.remove(T t)`, and `.has(T t)` are $O(n)$

- What should we do if the value we want to remove is in some range, $[a, b]$?

    - An array would work best here, because we have a contiguous range
    - Could use something like `myArray[b-a]` where `myArray[i-a]` indicates whether we have added an integer i into our storage

        * Initialize the entire array to zero to make it clear that it contains no elements
        * How would we perform `.add(T t)`, `.remove(T t)`, or `.has(T t)`?

- So an array works to tell us if something is in the set, because we can use the value at each index as a true or false boolean that tells us whether we have the element at the index

- What we would like *in general* is a mapping from any `Object` to some small manageable integer

    - We call this mapping a *hashing*

Terminology

- *Hash code*: the integer computer for any object

- *Hash function*: the function that computes a hash code for any object

- *Hash table*

    - Storage of objects based on their hash codes
        * Use a has code to perform a fast table lookup
    - Looks a lot like an array or a linked list
    - Usually we do not keep duplicates – duplicate elements get mapped to the same place in the hash table

Hash Table Example
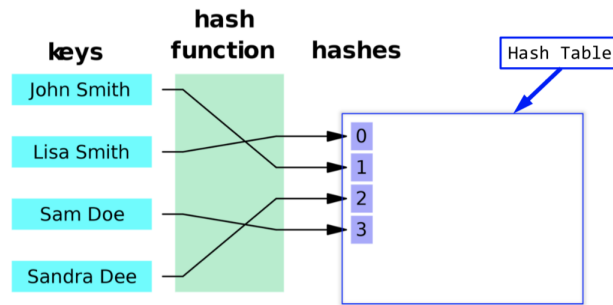
Hash Table

- Store objects in an array in a retrievable way

- A simplified view of `.add(T t)`:

    - Compute the integer hash code `xhc` from `x`
    - Put `x` in an array `hta` at the index `xhc`

- Things to consider

    - How do you compute `xhc`? Where should that code exist?
    - What if `xhc` is too big (it's larger than the length of the hash table)?
    - What if the current index is occupied?
        * We call this a *hash collision*

Roadmap

- Basic ideas of hashing

- Hash code computation

- Using hash codes

    - Bounding the range
    - Hash table collision resolution and other strategies

Hash Code

- An integer computed for an object

- Every object in Java has a `.hashCode()` method

    - `int hc = thing.hashCode();`
    - **Why?**
    - **How are hash codes computed by default?**
    - Official documentation

Overriding `.hashCode()`

- We can override `.hashCode()` to do something special on a per-class basis

    - For your own classes, override the default `.hashCode()`
    - Compute the hash based on the internal data of an object
    - Follow the hash contract

    ```java
    public class Student {
        Integer gnumber;
        String name;
        Department dept;

        // Other methods omitted

        @Override
        public int hashCode() { return gnumber.hashCode(); }
    }
    ```

Hash Contract

- `x.equals(y)` $\implies$ `x.hashCode() == y.hashCode()`

    - However, *different objects can have the same hash code*

- We can revisit our code snippet from above and make the hash more resilient

    ```java
    public class Student {
        Integer gnumber;
        String name;
        Department dept;

        // Other methods omitted

        @Override
        public int hashCode() { return gnumber.hashCode() + dept.hashCode(); }
    }
    ```

Picking a Good Hash Function

- Adhere to the *Hash Contract*

    - If $x = y$ then they must have the same hash

- *Distribute* different objects "fairly" across the integers

    - We get reduced collision as a benefit of this

- Compute the hash code as *quickly* as possible

    - Since we'll be doing a lot of hashing, we can greatly speed up the hash table with a fast hash function

- *Note*: We can't usually get all three of these, so we have to decide which, based on the problem, need to be prioritized

Hash Table (a Simplified View)

- Store objects in an array `hta` in a retrievable way
- `.add(T t)`: put the object `t` in the hash table
    - Compute the hash code `thc` from `t`
    - Use `thc` as the index of `hta` in which we will store `t`
- `.has(T t)`: check if `t` is in the hash table
    - Compute the hash code `thc` from `t`
    - `return t.equals(hta[thc]);`
- `.remove(T t)`: delete `t` from the hash table
    - Compute the hash code `thc` from `t`
    - `if has(t) { hta[thc] = null;}`

Hash Table: Issues

```java
class HashSet<T> {
    T hta[];
    int size;

    void add(T x) {
        int xhc = x.hashCode();
        // What if xhc is out of bounds?
        // What if this entry is already occupied?
        hta[xhc] = x;
        size++;
    }

    boolean has(T x) {
        int xhc = x.hashCode();
        return x.equals(this.hta[xhc]);
    }
}
```

Hash Code Processing

- Modulo (remainder) is the typical way to bound the hash code based on the hash table length:

```java
int n = hta.length;
hta[abs(xhc) % n] = x;
```

- For math-related reasons, it's usually best to make n a larger *prime* number
    - We do this so that multiples of the same number modulo n are less likely to collide

```
30 % 17 = 13                    30 % 9 = 3
300 % 17 = 11                   300 % 9 = 3
3000 % 17 = 8                   3000 % 9 = 3
30000 % 17 = 12                 30000 % 9 = 3
```

- Now we have bounding issues resolved, but we've shortened the range
    - It's more likely now that we'll have different objects mapping to the same entry

Hash Table Collisions

- Motivation

    - Put `x` in a table at `hta[xhc]`
    - **Problem**: what if `hta[xhc]` is occupied?
    - **Answer**: find some other storage for `x`

- Common approaches

   - Separate chaining
    - Open addressing

Separate Chaining

- Something already there?

    - Expand that single entry into an internal data structure
        * This way we can accommodate multiple objects which have the same hash code
        * We can also grow this structure if we get more collisions
    - We have a data structure for this... what is it?
        * A linked list (or an array list)
    - **What's the worst case complexity?**
        * **How can we avoid that case?**

Separate Chaining Examples

Example 1

```
String [] sa1 = new String[] {
    "Chris",
    "Sam",
    "Beth",
    "Dan"
};
SeparateChainHS<String> h = new SeparateChainHS<String>(11)
for(String s : sa1) {
    h.add(s);
}
// String("Chris").hashCode() % 11=65087095 % 11 = 7
// String("Sam").hashCode() % 11=82879 % 11 = 5
// String("Beth").hashCode() % 11=68465 % 11 = 1
// String("Dan").hashCode() % 11=2066967 % 11 = 1
```
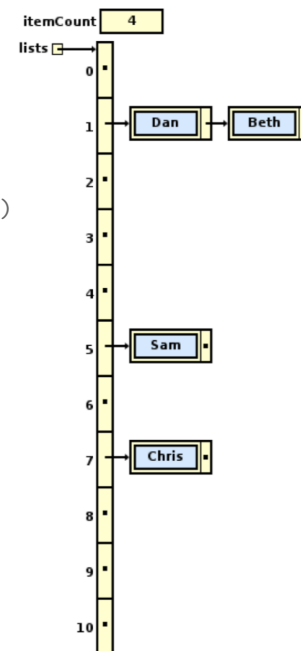

- Every table entry is a linked list

    - `hta[i]` stores all the values mapping to index `i`

- Example:
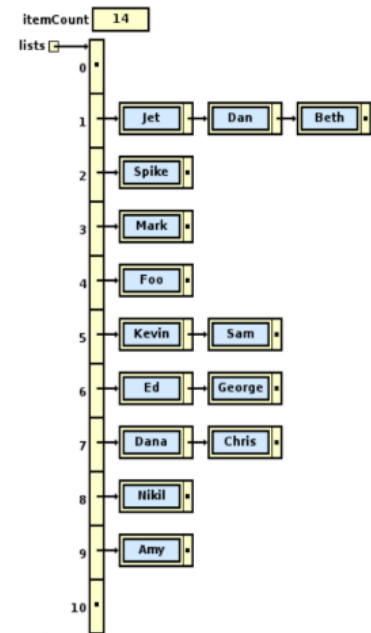 - Table length = 11
  - Number of items = 4
  - Load:
  $$\frac{\text{item count}}{\text{hash table length}} = \frac{4}{11} \approx 0.\overline{36}$$

Example 2

- Table length = 11

- Number of items = 4

- Load:
  $$\frac{\text{item count}}{\text{hash table length}} = \frac{14}{11}$$
  $$\frac{14}{11} = \approx 1.\overline{27}$$

Discussion

- Why use an array of linked lists?

- How can we calculate a good load factor?

Separate Chaining Analysis

- `.add(T t)` is $O(1)$ assuming adding to a list is $O(1)$ and that duplicates are allowed

  - If duplicates aren't allowed, the time jumps to $O(n)$ since we need to search the chain to make sure that we don't already have the element in the chain

- `.remove(T t)`/`.contains(T t)`

  - The run time depends on the number of things in each list

  - `.remove(T t)`/`.contains(T t)` must potentially look through all elements in the longest chain in the hash table to see if `t` is present
    * The average case is $O(\text{average chain length}) = O(\texttt{itemCount/tableLength}) = O(\text{load})$
    * The worst case time is $O(\texttt{itemCount})$
    * How do we avoid the worst case?

Separate Chaining is Viable in Practice

- It's relatively simple to implement (see Weiss Fig. 20.20)

- It's reasonably efficient

- Java's built-in hash tables use it

  - `java.util.HashSet`, `java.util.HashMap`, and `java.util.Hashtable` all use separate chaining