CS 61B Spring 2026

Hashing, Tries

Discussion 11: April 06, 2026

1 Summary

Hash Tables are used as backing data structures for `Set`s and `Map`s, and, in the best case, allow for constant-time `add` and `contains` operations. In memory, we construct them as an array with a "separate chain" at each index, such as a linked list. Often, we refer to these array indices as **buckets**. N is used to denote the number of items in the hash table, and M is used to denote the number of buckets.

- **Add**: Take the object's hash code, apply modulus by M. Search the corresponding bucket. If the bucket does not contain the object, add it to the end of the separate chain. Runs in $\Theta(1)$ assuming **good spread** (see below).
- **Contains**: Take the object's hash code, apply modulus by M. Search the corresponding bucket. If the bucket contains the object, return `True`. Else, return `False`. Runs in $\Theta(1)$ assuming **good spread** (see below).
- **Resizing**: We define our hash table's **load factor** as the ratio N/M. We also define some load factor that will trigger a resize, and a **scaling factor** by which the array's size will be multiplied. After each `add` operation, we check N/M, and if our target load factor is reached, we initiate a resize. During a resize, we rehash every item in the hash table, which may change its corresponding bucket.
- **Good Spread vs Bad Spread:** To reduce the runtime of `add` and `contains` we want items to be evenly distributed within our hash table. `hashCode` implementations that result in a relatively even distribution are said to have "good spread". `hashCode` implementations that result in uneven distribution are said to have "bad spread".

Tries are trees specialized for language tasks. Each node represents a character, and "marked" nodes denote the ends of words.

- **longestPrefixOf(String s):** Follow the trie until the letters no longer match, keeping track of the most recent "marked" node. Runs in $O(L)$, where L is the length of the longest key.
- **keysWithPrefix(String s):** Follow the trie until the end of the prefix, and return all words indicated by marked nodes below that node. Runs in $O(L)$, where L is the length of the longest key.

2 A Side of Hash Browns

We want to map food items to their yumminess. We want to be able to find this information in constant time, so we've decided to use Java's built-in **`HashMap`** class! Here, the key is an **`String`** representing the food item and the value is an **`int`** yumminess rating.

For simplicity, let's say that here a **`String`**'s hashcode is the first letter's position in the alphabet (A = 0, B = 1... Z = 25). For example, the **`String "Hashbrowns"`** starts with "H", and "H" is 7th letter in the alphabet (0 indexed), so the hashCode would be 7. Note that in reality, a **`String`** has a much more complicated **`hashCode`** implementation.

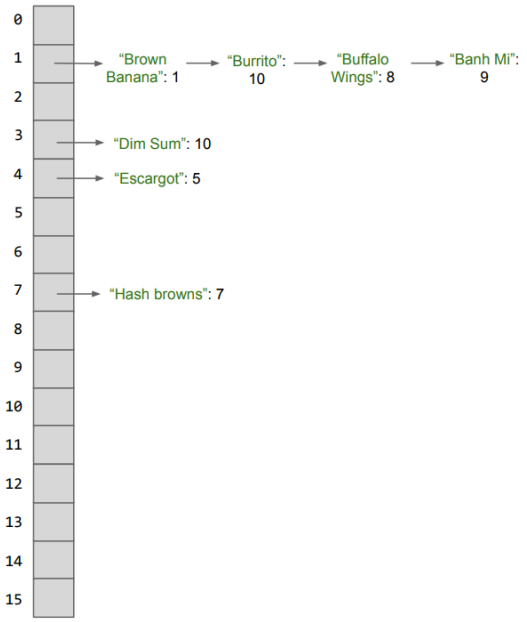
Our **`HashMap`** will compute the index as the key's hashcode value modulo the number of buckets in our **`HashMap`**. Assume the initial size is 4 buckets, and we double the size of our **`HashMap`** as soon as the load factor reaches 3/4. If we try to put in a duplicate key, simply replace the value associated with that key with the new value.

(a) Draw what the **`HashMap`** would look like after the following operations.

```
HashMap<String, Integer> hm = new HashMap<>();
hm.put("Hashbrowns", 7);
hm.put("Dim sum", 10);
hm.put("Escargot", 5);
hm.put("Brown bananas", 1);
hm.put("Burritos", 2);
hm.put("Buffalo wings", 8);
hm.put("Banh mi", 9);
hm.put("Burritos", 10);
```

Solution:

Note that we resize from 4 to 8 when adding escargot (because we have 3 items and 4 buckets) and then from 8 to 16 when adding buffalo wings (because we have 6 items and 8 buckets).



(b) Do you see a potential problem here with the behavior of our **`HashMap`**? How could we solve this?

Solution:

Here, adding a bunch of food items that start with the letter "B" result in one bucket with a lot of items. No matter how many times we resize, our current `hashCode` will result in this problem! Imagine if we added in 100 more items that started with the letter b. Though we would resize and keep our load factor low, it wouldn't change the fact that our operations will now be slow (hint: linear time) because now we essentially have to iterate over a linked list with pretty much all the items in the hashMap to do things like `get("Burrito")`, for example.

A solution to this would be to have a better `hashCode` implementation. A better implementation would distribute the `String`s more randomly and evenly. While knowing how to write such a `hashCode` is difficult and out of scope for this class, you can look at the real `hashCode` implementation for Java `String`s if you're curious!

3 Hashing

(a) Here are five potential implementations of the `Integer` class's `hashCode()` method. Categorize each as having (1) Good spread, (2) Bad spread, and (3) No spread.

For the 2nd implementation, note that ***`intValue()`*** *will return that* ***`Integer`****'s number value as an* ***`int`****.*

```
public int hashCode() {
    return -1;
}
```

Solution: No spread

```
public int hashCode() {
    return super.hashCode(); // Object's hashCode() is based on memory location
}
```

Solution: Good spread

The exact reason is somewhat out of scope of this course, but Java's `System.identityHashCode` does bit manipulation to ensure good spread.

Reasoning that memory addresses are roughly random and should, as a result, have good spread is enough.

```
public int hashCode() {
    return intValue() * intValue() * 2;
}
```

Solution: Bad spread Integers that share the same absolute values will collide (for example, `x = 5` and `x = -5` will have the same hash code). Additionally, if the hash table has an even number of buckets, only half of the buckets will ever be used. A better hash function would be to just return the `intValue` itself.

```
public int hashCode() {
    return (int) (new Date()).getTime(); // returns the current time as an int
}
```

Solution: Good spread

```
public int hashCode() {
    return intValue() + 3;
}
```

Solution: Good spread

(b) For each of the following questions, answer **Always**, **Sometimes**, or **Never**.

1. If you were able to modify a key that has been inserted into a `HashMap` would you be able to retrieve that entry again later? For example, let us suppose you were mapping ID numbers to student names, and you did a `put(303, "Dawn")` operation. Now, let us suppose we somehow went to that item in our `HashMap` and manually changed the key to be 304. If we later do `get(304)`, will we be able to find and return `"Dawn"`? Explain.

Solution: Sometimes. If the `hashCode` for the key happens to change as a result of the modification, then we won't be able to retrieve the entry in our hashtable (unless we were to recompute which bucket the new key would belong to). Changing the key can potentially (and likely) change which bucket a key would now be indexed to.

In our example, let us suppose the `hashCode` for a key was just it's integer value, and the bucket was calculated as that number modulo the number of buckets. Let's say we had 10 buckets. In this scenario, 303 would be mapped to bucket 3, and 304 would be mapped to bucket 4. So when we `put(303, "Dawn")`, this item goes to bucket 3. Then we change the key to be 304, but don't change anything else, so this item remains in bucket 3. Now, let us suppose we later do `get(304)`. We will compute which bucket the key 304 would correspond to, which would be bucket 4. We look in bucket 4 and don't see the time there, so we cannot successfully return the item.

2. When you modify a value that has been inserted into a `HashMap` will you be able to retrieve that entry again? For example, in the above scenario, suppose we first inserted `put(303, "Dawn")` and then changed that item's value from `"Dawn"` to `"Michelle"`. If we later do `get(303)`, will we be able to find and return `"Michelle"`? Explain.

   Solution: Always. The bucket index for an entry in a `HashMap` is decided by the key, not the value. Mutating the value does not affect the lookup procedure.

   In our example this would work, because when we do `get(303)` we will still go to the correct bucket.

4 Buggy Hash

The following classes may contain a bug in one of its methods. Identify those errors and briefly explain why they are incorrect and in which situations would the bug cause problems.

(a) The `Timezone` class below:

```
class Timezone {
    String timeZone; // "PST", "EST" etc.
    boolean dayLight;
    String location;
    ...
    public int currentTime() {
        // return the current time in that time zone
    }

    public int hashCode() {
        return currentTime();
    }

    public boolean equals(Object o) {
        Timezone tz = (Timezone) o;
        return tz.timeZone.equals(timeZone);
    }
}
```

Solution: Although equal objects will have the same hashcode, but the problem here is that `hashCode()` is not deterministic. This may result in weird behaviors (e.g. the element getting lost) when we try to put or access elements.

(b) The `Course` class below:

```
class Course {
    int courseCode;
    int yearOffered;
    String[] staff;
    ...
    public int hashCode() {
        return yearOffered + courseCode;
    }

    public boolean equals(Object o) {
        Course c = (Course) o;
        return c.courseCode == courseCode;
    }
}
```

Solution: The problem with this `hashCode()` is that not all equal objects have the same hashcode. This may produce unexpected behavior, e.g. multiple "equal" objects may be exist in different buckets in the HashMap, the `containsKey` operation may return false, etc. One key thing to remember is that when we override the `equals()` method, we have to also override the `hashCode()` method to ensure equal objects have the same hashCode.

5 Wordsearch

Given an N by N wordsearch and N words, devise an algorithm (using pseudocode or describe it in plain English) to solve the wordsearch in $O(N^3)$. For simplicity, assume no word is contained within another, i.e., if the word **"bear"** is given, **"be"** wouldn't also be given.

If you are unfamiliar with wordsearches or want to gain some wordsearch solving intuition, see below for an example wordsearch.

Example Wordsearch:

S F T A T R D V R G K E V I N
A T S K L N X F I H D P X H Z
C N E D X A J Z G U N A I R U
J Y I L C V A N E S S A V P O
A B A R L K F J Q U S Y H I C
V Z U I U A T Q K D A A G R D
F S P E I D S T A A S N M Y N
W C T T S S H Q T W H N G A U
S I W H P E A A E N L I T N V
T Y I A G L D B R K E Y T Y K
A L N N J A J G E T Y A P E A
C X F I K I M U S L I T Y P R
E M U A M N T M A Z X A H U E
Y R A E K E Y W I K O Y O P N
R B C A Q J V Q I A C R O E F

Anirudh	Anniyat	Vanessa	Ryan
Ashley	Elaine	Isabel	
David	Stella	Karen	
Ethan	Kevin	Teresa	
Stacey	Dawn		

Hint: Add the words to a **Trie**, and you may find the `longestPrefixOf` operation helpful. Recall that `longestPrefixOf` accepts a `String key` and returns the longest prefix of `key` that exists in the **Trie**, or `null` if no prefix exists.

Solution:
Algorithm: Begin by adding all the words we are querying for into a **Trie**. Next, we will iterate through each letter in the wordsearch and see if any words **start** with that letter. For a word to start with a given letter, note that it can go in one of eight directions — N, NE, E, SE, S, SW, W, NW.

Looking at each direction, we will check if the string going in that direction has a prefix that exists in our **Trie**, which we can do using `longestPrefixOf`. Note that words are not nested inside of others, so **at most** one word can start from a given letter in a given direction. As such, if `longestPrefixOf` returns a word, we know it is the only word that goes in that direction from that letter.

For instance, if we are at the letter "K" in the middle of the top row of the wordsearch above and are considering the direction east, we would want to see if the string `"KEVEN"` has a prefix that exists in the given wordsearch. To efficiently perform this query, we call `longestPrefixOf("KEVIN")`, which, in this case, returns `"KEVIN"`, and we proceed by removing `"KEVIN"` from our **Trie** to signal that we found the word `"KEVIN"`.

We will repeat this process until all the words have been found, i.e. when the **Trie** is empty. Finally, note that this is a very open-ended problem, so this is one of **many** possible solutions.

Runtime: We look at N^2 letters. At each letter, we execute eight calls to `longestPrefixOf` which runs in time linear to the length of the inputted string, which can be of at most length N, since that is the height and width of the wordsearch. Thus, if we perform on the order of N work per letter and we look at N^2 letters, the runtime is $O(N^3)$.