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);
```

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

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;
}
```

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

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

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

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

(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.

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.

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);
    }
}
```

(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;
    }
}
```

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.