CS 61B Spring 2026

Hashing, Tries

Exam-Level 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 Hashing: Gone Crazy

For this question, use the following `TA` class for reference.

```
public class TA {
    int semester;
    String name;
    TA(String name, int semester) {
        this.name = name;
        this.semester = semester;
    }
    @Override
    public boolean equals(Object o) {
        TA other = (TA) o;
        return other.name.charAt(0) == this.name.charAt(0);
    }
    @Override
    public int hashCode() { return semester; }
}
```

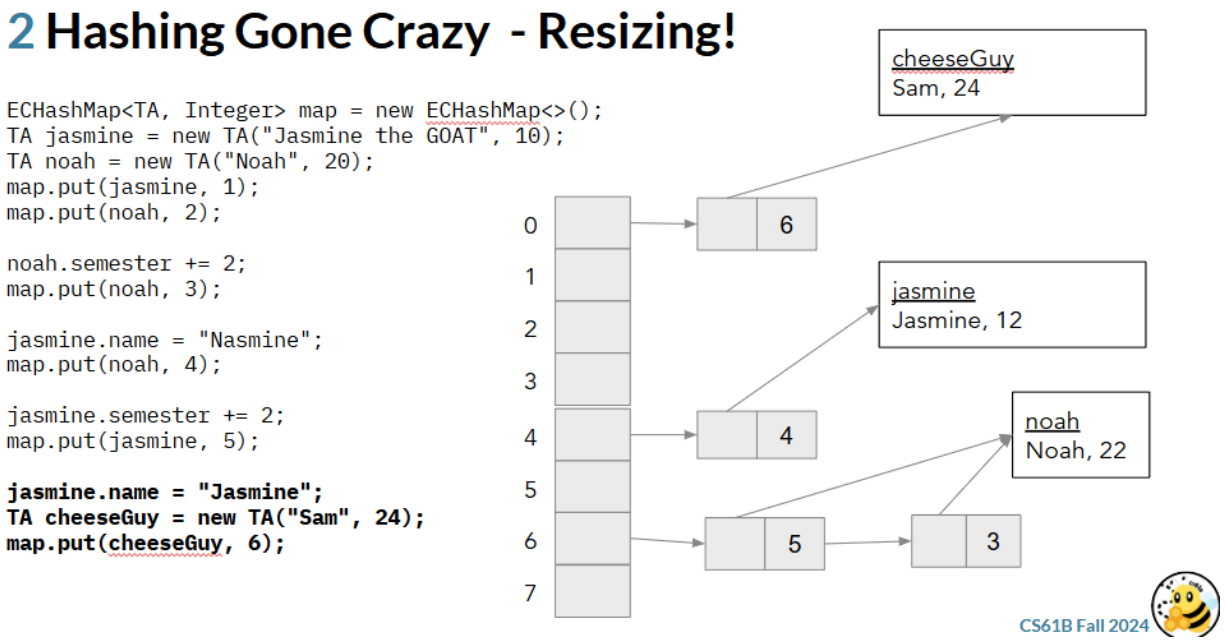
Assume that the `ECHashMap` is a `HashMap` implemented with **external chaining** as depicted in lecture. The `ECHashMap` instance begins with 4 buckets.

Resizing Behavior If an insertion causes the load factor to reach or exceed 1, we resize by doubling the number of buckets. During resizing, we traverse the linked list that correspond to bucket 0 to rehash items one by one, and then traverse bucket 1, bucket 2, and so on. Duplicates are **not** checked when rehashing into new buckets.

Draw the contents of map after the executing the insertions below:

```
ECHashMap<TA, Integer> map = new ECHashMap<>();
TA jasmine = new TA("Jasmine the GOAT", 10);
TA noah = new TA("Noah", 20);
map.put(jasmine, 1);
map.put(noah, 2);

noah.semester += 2;
map.put(noah, 3);

jasmine.name = "Nasmine";
map.put(noah, 4);

jasmine.semester += 2;
map.put(jasmine, 5);

jasmine.name = "Jasmine";
TA cheeseguy = new TA("Sam", 24);
map.put(cheeseguy, 6);
```

Solution:



Explanation:

Line 4: `jasmine` has `semester` value `10`. `10 % 4 = 2`, so `jasmine` is placed in bucket 2 with value 1.
`0: [], 1: [], 2: [(jasmine, 1)], 3: []`

Line 5: `noah` is placed in bucket 0 with value 2.
`0: [(noah, 2)], 1: [], 2: [(jasmine, 1)], 3: []`

Line 7: Increasing the `semester` value of `noah` does *not* cause it to be rehashed! (This is why modifying objects in a Hashmap is dangerous - it can change the hashcode of your object and make it impossible to find which bucket it belongs to).

Line 8: `noah` now has `semester` 4, so bucket 2 also has a node pointing to `noah`, with value 3. (Note that the two `noahs` refer to the same object).
`0: [(noah, 2)], 1: [], 2: [(jasmine, 1), (noah, 3)], 3: []`

Line 11, 12: `noah` with `semester` 22 hashes to bucket 2. However, since we have changed `jasmine`'s name to be "`Nasmine`", `noah.equals(jasmine)` returns `true`. Since we are hashing a key that is already present in the dictionary according to `.equals`, we replace `jasmine`'s old value with the new value, 4.
`0: [(noah, 2)], 1: [], 2: [(jasmine, 4), (noah, 3)], 3: []`

Line 13, 14: `jasmine` with `semester` 12 hashes to bucket 0. However, since we have changed `jasmine`'s name to be "Nasmine", `jasmine.equals(noah)` returns `true`. Since we are hashing a key that is already present in the dictionary according to `.equals`, we replace `noah`'s old value with the new value, 5.
`0: [(noah, 5)], 1: [], 2: [(jasmine, 4), (noah, 3)], 3: []`

Line 16, 17, 18: `cheeseGuy` hashes to bucket 0. `cheeseGuy.equals(noah)` returns `false`, so we add a new node after `noah` with value 6.
`0: [(noah, 5), (cheeseGuy, 6)], 1: [], 2: [(jasmine, 4), (noah, 3)], 3: []`

Resizing: The load factor reaches 1, and we resize to 8 buckets. We rehash the elements in the order they were inserted. Notice that duplicates are not checked when rehashing into new buckets, therefore, the `noah` object is inserted twice.

3 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.

4 Wordsearch

Fill in the **`longestPrefixOf(String word)`** method below such that it returns the longest prefix of **`word`** that is also a prefix of a key in the trie.

For example, if a **`TrieSet t`** contains keys **`{"cryst", "tries", "cr"}`**, then **`t.longestPrefixOf("crystal")`** returns **`"cryst"`** and **`t.longestPrefixOf("crys")`** returns **`"crys"`**.

The code uses the **`StringBuilder`** class to build strings character-by-character. To add a character to the end of the **`StringBuilder`**, use the **`append(char c)`** method. Once all characters have been appended, the resulting String is returned by the **`toString()`** method.

```
StringBuilder sb = new StringBuilder();
sb.append('a');
sb.append('b');
System.out.println(sb.toString()); // "ab"
```

```
public class TrieSet {
    private Node root;
    private class Node {
        boolean isKey;
        Map<Character, Node> map;
        private Node() {
            isKey = false;
            map = new HashMap<>();
        }
    }

    public String longestPrefixOf(String word) {
        int n = word.length();
        StringBuilder prefix = new StringBuilder();
        Node curr = _____________________;
        for (_____________________________________) {
            ____________________________________________
            ____________________________________________
            ____________________________________________
            ____________________________________________
            ____________________________________________
            ____________________________________________
        }
        return _____________________________________
    }
}
```

Solution:

```
public String longestPrefixOf(String word) {
    int n = word.length();
    StringBuilder prefix = new StringBuilder();
    Node curr = root;
    for (int i = 0; i < n; i++) {
        char c = word.charAt(i);
        if (!curr.map.containsKey(c)) {
            break;
        }
        curr = curr.map.get(c);
        prefix.append(c);
    }
    return prefix.toString();
}
```