CS 61B
Fall 2026

Hashing II, Tries, Software Engineering

Discussion 11: November 9, 2026

1 Software Engineering

In Aniruth's guest lecture on Friday, he talked about how the best classes are *deep*: they hide a lot of complexity, and the interface to users (i.e., what the users need to know in order to use the class) is comparatively small and simple. Let's reflect on how deep one of the classes that you wrote for project 4C/D is. Open up your code on your laptop (really!), and pick one class that you implemented as an example.

What's the name of the class? _____

(a) What's one piece of information or complexity that is completely hidden within the class, so users don't need to know about it. Possible examples include "the relationships between different words and how they're stored" or "how we keep track of the word frequency for each year."

Whatever you wrote above is an example of hiding complexity within a class, which is great! That class is helping its users, by shielding them from complexity that the user doesn't need to know about.

(b) Next, identify something in that class that's not obvious from method signatures, but is something that a user needs to know. For example, do you have a set of somewhat obscure methods that must all be called in order to achieve some task?

At a minimum, this information should be described in a comment, to warn users about it. Is there any way to restructure your code to keep this from leaking out (the best solution!) or to make it obvious from the method signatures?

(c) As you were thinking about this and looking back at your code, you got a look at the variable and method names that you used. List the worst two variable and method names that you used in your code. Explain one to your group. Can anyone in your group come up with a better name?

2 Set the Date

Dawn is seriously concerned about how much money Ben spends going out to eat. She wants to create a set so that she can efficiently check whether or not Ben has gone out to eat on a particular day. To do so, she creates the Date class below.

```
public class Date {
  /* An integer from 1 to 12 (inclusive), representing the month of the year */
  public int month;
  /* An integer from 1 to 31 (inclusive), representing the day of the month */
  public int day;
  /* The numerical year */
  public int year;

  public Date(int month, int day, int year) { this.month = month;
                                              this.day = day; this.year = year; }

  public boolean equals(Object o) {
    if (o instanceof Date d) {
      return month == d.month && day == d.day && year == d.year;
    }
    return false;
  }

  public int hashCode() {
    /* TODO: implement hashCode */
    return -1;
  }
}
```

(a) Evaluate the performance of the **hashCode** implementations below. Write either "good spread", "bad spread", and briefly explain why. Also state whether hash table operations will work correctly (i.e. is **hashCode** consistent with **equals**?)

```
public int hashCode() {
  return this.month;
}
```

Bad spread. **month** is an integer from 1 to 12, and as such, many buckets will be left unused if we store many dates. However, operations will work correctly: equal **Date**s have the same month, so they'll always get the same hash code.

```
public int hashCode() {
  return 2 * this.day;
}
```

Still, bad spread. **day** covers a marginally larger range of integers (2 to 62), but all odd buckets will be left unused if the number of buckets is even. Operations still work correctly: equal **Date**s have the same day, so they'll produce the same hash code.

(b) Okay, looks like it's gonna be a little harder than Dawn thought. She proposes this **hashCode** implementation instead. Does it have good spread? Will hash table operations work correctly (is it valid)?

```
public int hashCode() {
  return (int)(Math.random() * 1000);
}
```

Since this hashCode is not deterministic, **contains** might not look in the bucket that actually contains our **Date**.

(c) Dawn remembers from lecture that, to solve this issue for lowercase English **String**s, we converted the string into a base-26 integer.

Fill in the **hashCode** method below, returning the date as a base-31 integer.

```
public int hashCode() {
  /* Dawn started collecting dates in 2026, so she
  wants to start the 'year' digit at 2025 */
  return (year - 2025) * 31*31 + month * 31
    + day * 1;
}
```

(d) Why do you think base-31 was chosen? Briefly explain.

There are a maximum of 31 days in a month, so, if we want our **Date**s to have 1-to-1 integer representations, we need a base of at least 31.

31 is also a prime number, which is great for reasons slightly beyond the scope of this class.

(e) Suppose that we didn't implement **hashCode()** at all. Fill in the blanks below to demonstrate a situation where **hashCode** and **equals** inconsistency results in an incorrect answer.

```
HashSet<Date> set = new HashSet<>();
Date d1 = new Date(1, 1, 2026);
Date d2 = new Date(1, 1, 2026);
set.add(d1);
System.out.println(d1.equals(d2)); // prints true
System.out.println(set.contains(d2)); // prints false, should be true
```

Fill in **d2** with the same values as **d1** so that **d1.equals(d2)** is **true**. However, **set.contains(d2)** might print **false**: without a **hashCode** override, **Date** inherits **Object.hashCode()**, which is based on memory address. Since **d1** and **d2** are different objects, they will likely have different hash codes and land in different buckets — so **contains** won't find **d1** when searching for **d2**, even though they are equal.

3 Longest Prefix

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

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();
        String prefixToReturn = "";
        StringBuilder prefixSoFar = "";
        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);
            prefixSoFar.append(c);
            if (curr.isKey) {
                prefixToReturn = prefixSoFar.toString();
            }

        }
        return prefixToReturn;
    }

}
```

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();
}
```