CS 61B Spring 2026

Re: Hashing, Re: Tries

Discussion 12: April 13, 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. Alternately, can bound as $O(|s|)$ where is the length of `s`.
- **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 $O(L + Z)$, where L is the length of the longest key and Z is the number of keys returned.

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?

```
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 **Strings**, 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 Separate Chaining

Suppose that we have a hash table implemented using separate chaining **with an LLRB in each bucket** instead of a Linked List. Assume that the hash table's key implements `Comparable`.

(a) Using this hash table,

The runtime of `contains` is, in the worst case:

Runtime: Θ ($\log N$)

The runtime of `contains` is, in the best case:

Runtime: Θ (1)

(b) Evaluate the following statements as **true** or **false**, and briefly explain why.

"One advantage of using an LLRB for buckets is that it becomes possible to efficiently iterate over the keys in ascending order"

False. While it becomes possible to iterate over keys efficiently in ascending order within buckets using a tree traversal, it is still difficult to iterate over all of the hash table's keys, as each bucket contains a distinct LLRB.

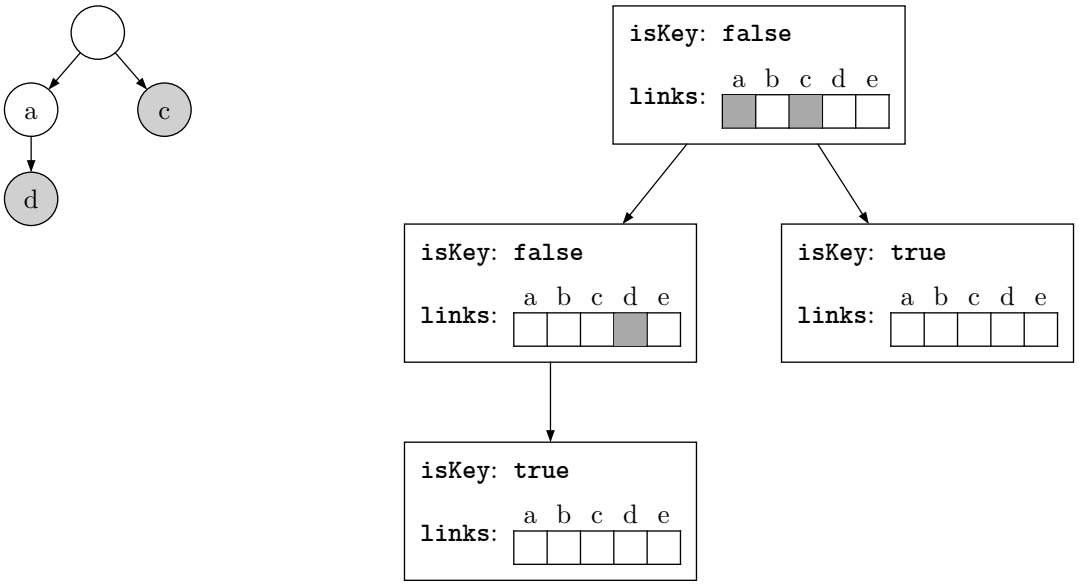
"Assuming items in the hash table have good spread, we expect that an LLRB bucket would yield significantly better performance for `contains` and `add` than using an `ArrayList`"

False. If the items have good spread, number of elements in each bucket should be constant relative to N, so an asymptotic speedup doesn't yield significantly better performance.

4 Trie Tradeoffs

In lecture, we discussed Tries implemented with **`DataIndexedCharMap`**s for child tracking. A **`DataIndexedCharMap`** is like an array indexed by **`char`**s, and takes up R space in memory, where R is the size of our alphabet. For example, if we're only tracking lowercase letters in the English alphabet, $R = 26$.

Below is an example of a trie with the alphabet **`{'a', 'b', 'c', 'd', 'e'}`** containing the strings **`"ad"`** and **`"c"`**, and its representation in memory.



(a) Give a theta bound on the asymptotic memory usage of a Trie using **`DataIndexedCharMap`**s in terms of N and R, where N is the number of nodes in our Trie and R is the size of our alphabet.

Memory usage: : $\Theta\ (\ RN\ )$

Wow! That's not ideal. Can we improve this at all?

(b) Propose an alternative data structure we could use for child tracking such that, in terms of R and L, where L is the length of our longest key, `contains` runs in $\Theta(L \log R)$ in the worst case.

Hint: We can compare characters by comparing their integer representations

Solution: Either a 2-3 tree or an LLRB is suitable here. These data structures both ensure bushiness, hence the $\log R$ part of the solution.

(c) Give a theta bound on the asymptotic memory usage of a Trie using your data structure of choice for child tracking in terms of N and R, where N is the number of nodes in our Trie and R is the size of our alphabet.

Memory usage: : $\Theta (N)$

(d) Propose a situation where you might want to use a `DataIndexedCharMap`-backed Trie and a situation where a Trie backed by your data structure of choice might make more sense.

Hint: Look at the runtime and memory bounds you calculated!

Solution: There are quite a few possible answers here, but notice that our memory usage for a `DataIndexedCharMap`-backed Trie depends on R. In situations where we have a huge alphabet (like if we're using Unicode mappings, for example), our memory usage will absolutely explode.

Alternatively, if we have a small alphabet, the better performance of `DataIndexedCharMap`s might be more appealing.