CS 61B Fall 2026

Hashing, Midterm Review

Discussion 10: November 2, 2026

1 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.

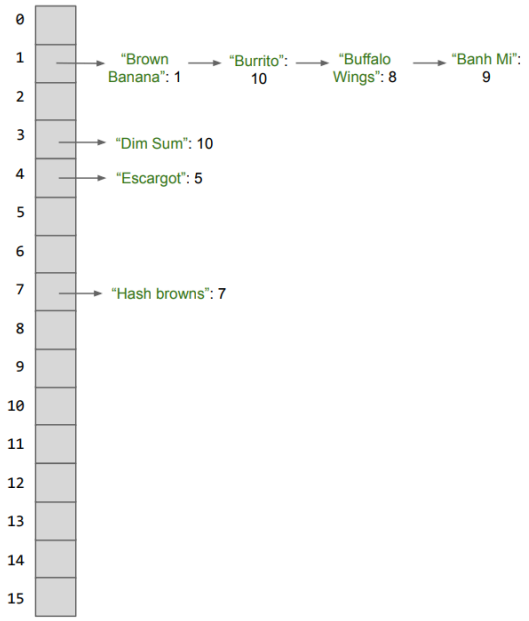
Our `HashMap` will compute the index as the key's hashcode value modulo the number of buckets in our `HashMap`. Assume the initial size is 4 buckets, and we double the size of our `HashMap` as soon as the load factor reaches 3/4. If we try to put in a duplicate key, simply replace the value associated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

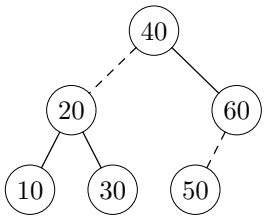
Here, adding a bunch of food items that start with the letter "B" result in one bucket with a lot of items. No matter how many times we resize, our current `hashCode` will result in this problem! Imagine if we added in 100 more items that started with the letter b. Though we would resize and keep our 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!

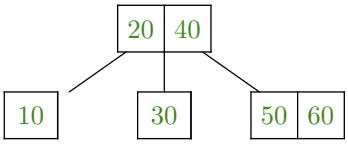
2 LLRBs

from Fall 2022 Midterm 2

Consider the below integer LLRB. Like typical LLRBs, assume all values are unique.



(a) Draw the corresponding 2-3 tree.



(b) Give all integers which, when inserted into the LLRB, result in a single `rotateLeft` operation and no further cascading operations (i.e. no `rotateLeft`s, `rotateRight`s or `colorFlip`s). If this is not possible, write "impossible".

If it is possible, use **inclusive square bracket notation**, e.g. if your answer is "`[62, 64]` and `[66, 68]`", this is equivalent to saying 62, 63, 64, 66, 67, 68.

`[11, 19]` and `[31, 39]`

We `rotateLeft` when there is a right-leaning red node with no left-leaning red neighbor. To prevent cascading operations, we must also do this on a black node (or else we'd trigger a `rotateRight`). So, we must add a right-leaning red child to `10` or `30`, as they are the only nodes in our tree that fit these criteria. To do this, we can add an integer from 11 to 19 (inclusive) or 31 to 39 (inclusive).

(c) Give all integers which, when inserted into the LLRB results in a single `rotateRight` operation and no further cascading operations (i.e. no `rotateLeft`s, `rotateRight`s or `colorFlip`s). If this is not possible, write "impossible". If it is possible, use inclusive square bracket notation.

Impossible

`rotateRight` always triggers a `colorFlip` according to typical LLRB invariants, so this is impossible.

(d) At most, how many additional inserts can be performed into the LLRB above without triggering any `colorFlip` operations? Do not count the operation that results in the first color flip.

(b) If we insert a right child on 10 and then 30, it will trigger two `rotateLeft`s. The next insertion, no matter where, will trigger a `colorFlip`.

3 I Am Speed

For each code block below, fill in the blank(s) so that the function has the desired runtime. If the answer is impossible, just write "impossible" in the blank. Assume that `System.out.println` runs in constant time. You may use Java's `Math.pow(x, y)` to raise `x` to the power of `y`.

(a) Desired Runtime: $\Theta(N)$

```
public static void f1(int N) {
    for (int i = 1; i < N; i += 1){
        System.out.println("hi Selina");
    }
}
```

Note the solution could be `i += C`, where `C` is some constant independent of `N`. This is because even if we did for example, `i += 10`, we would do $\frac{N}{10}$ work in total, which is still $\Theta(N)$.

(b) Desired Runtime: $\Theta(\log N)$

```
public static void f2(int N) {
    for (int i = 1; i < N; i *= C) {
        System.out.println("howdy Caitlin");
    }
}
```

Here, the solution could be `i *= C`, where `C` is some constant independent of `N`. This is because even if we did for example, `i *= 5`, we would do $\log_5 N$ work in total, and in general $\log_i N$ work, which is still $O(\log n)$.

(c) Desired Runtime: $\Theta(1)$

```
public static void f3(int N) {
    for (int i = 1; i < 1000; i += 1) {
        System.out.println("hello Dawn");
    }
}
```

Again, the solution is actually just `i < C`, where `C` is some constant independent of the input `N`. Any other syntactically valid constant would work.

(d) Desired Runtime: $\Theta(2^N)$

This one is tricky! *Hint: think about the dominating term in* $1 + 2 + 4 + 8 + ... + f(N)$

```
public static void f4(int N) {
    for (int i = 1; i < Math.pow(2, N); i *= 2) {
        for (int j = 0; j < i; j += 1) {
            System.out.println("what's up Alyssa");
        }
    }
}
```

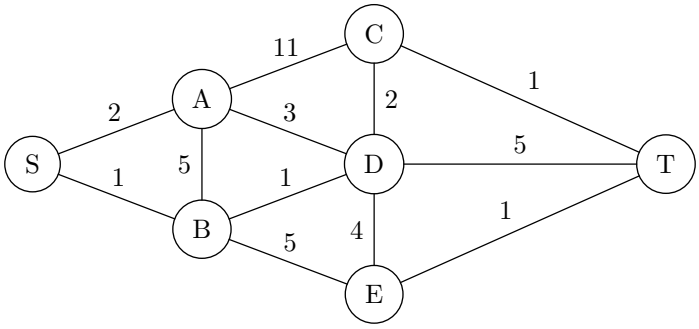
4 Lost in the Ceryneian H.I.N.D.

from Spring 2024 Final

After being repeatedly hacked in past exams, you've finally tracked down the hackers responsible as members of the Ceryneian H.I.N.D. With the assistance of an investigator (codenamed Heracles), you have managed to infiltrate their internal network. Help Heracles catch the Ceryneian H.I.N.D.

The internal network is a **weighted, undirected graph**, with start vertex S and target vertex T. You want to delete one edge from the graph, such that when deleted, the shortest path length from S to T is increased by as much as possible.

For example, consider the graph below:



Before deleting an edge, the shortest path from S to T has length 5. If we delete edge BS, the shortest path from S to T will instead have length 8; deleting any other edge will result in a shortest path from S to T that is shorter than 8. Thus, we should delete edge BS.

Design an algorithm (in English or pseudocode) that, given a graph, returns the edge that should be deleted. If multiple edges yield the same maximal shortest-path length, your algorithm may return any of them. You may assume that:

- All edges have positive integer edge weights.
- $S \neq T$
- The input graph is connected, and cannot be disconnected by removing any one edge.
- You have access to a working black-box implementation of Dijkstra's algorithm, which runs in $\Theta(E \log V)$ time.

For credit, your algorithm must run in $O(VE \log(V))$ time, where V and E are the number of vertices and edges in the graph, respectively.

Hint: Try to devise a naive solution first, then optimize that solution down to the specified runtime bound.

First, run Dijkstra's algorithm on the unmodified graph to get the shortest path from S to T. Then, for every edge on the shortest path from S to T, run Dijkstra's algorithm on the graph with just that edge removed, and record the resulting distance from S to T. Return the edge that, when removed, resulted in the largest distance from S to T.

Sometimes, when solving algorithm-design questions, it helps to come up with a naive, slow algorithm first, and then work on optimizing it. The naive solution to this problem would be to check every edge. In other words, run Dijkstra's algorithm E times, with a different edge removed each time, and return the edge that, when removed, resulted in the largest distance from S to T.

However, this naive algorithm is too slow. Each run of Dijkstra's algorithm takes $\Theta(E \log V)$. time, and we are running Dijkstra's algorithm E times, for a total runtime of $\Theta(E^2 \log V)$.

The key realization to optimize this solution is: If an edge is not on the original shortest path from S to T, then removing that edge will not increase the shortest path length at all, since we can just use the same original shortest path. Therefore, we only have to check edges that were on the original shortest path from S to T.

Finally, to finish the problem, we can verify that our optimized solution meets the required runtime bound. Since all edge weights are positive, the shortest path from S to T will not contain cycles. (Cycles would just increase the length of the shortest path.) Therefore, the shortest path has most $V - 1$ edges (which would occur if the shortest path passed through every single vertex). The shortest path would not pass through a vertex twice, since that would create a cycle.

We have to run Dijkstra's algorithm once on the unmodified graph to get the original shortest path from S to T. Then, we run Dijkstra's algorithm V $-$ 1 more times, to check each edge on the shortest path from S to T. This means we run Dijkstra's algorithm a total of V times. Each run of Dijkstra's algorithm takes $\Theta(E \log V)$, for a total runtime of $\Theta(VE \log V)$. This meets the $O(VE \log V)$ bound specified in the question, and we are done.