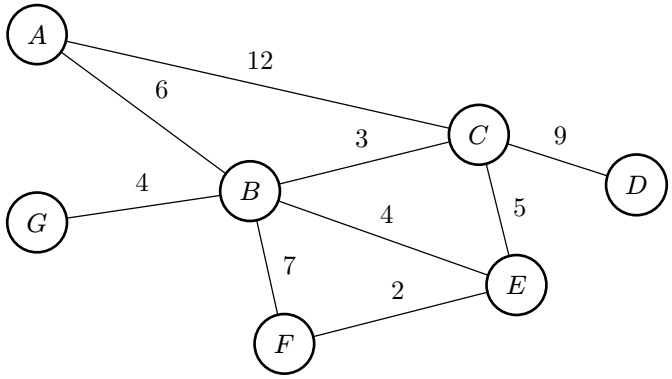
CS 61B Fall 2025

MSTs, DAGs, and Hashing

Discussion 09: October 27, 2025

1 Introduction to MSTs



(a) For the graph above, list the edges in the order they're added to the MST by Kruskal's and Prim's algorithm. Assume Prim's algorithm starts at vertex A. Assume ties are broken in alphabetical order. Denote each edge as a pair of vertices (e.g. AB is the edge from A to B).

Prim's algorithm order:

Kruskal's algorithm order:

Solution: Prim's algorithm order: **AB, BC, BE, EF, BG, CD**
Kruskal's algorithm order: **EF, BC, BE, BG, AB, CD**

(b) True/False: Adding 1 to the smallest edge of a graph G with unique edge weights must change the total weight of its MST.

Solution: True, either this smallest edge (now with weight +1) is included, or this smallest edge is not included and some larger edge takes its place since there was no other edge of equal weight. Either way, the total weight increases.

(c) True/False: If all the weights in an MST are unique, there is only one possible MST.

Solution: True, the cut property states that the minimum weight edge in a cut must be in the MST. Since all weights are unique, the minimum weight edge is always unique, so there is only one possible MST.

(d) True/False: The shortest path from vertex u to vertex v in a graph G is the same as the shortest path from u to v using only edges in T, where T is the MST of G.

Solution: False, consider vertices C and E in the graph above. The shortest path between C and E uses the edge CE, but it is not part of the MST of the graph.

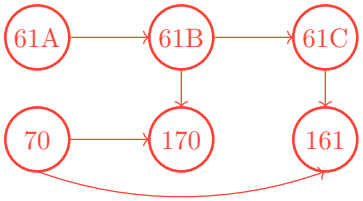
2 Class Enrollment

You're planning your CS classes for the upcoming semesters, but it's hard to keep track of all the prerequisites! Let's figure out a valid ordering of the classes you're interested in. A valid ordering is an ordering of classes such that every prerequisite of a class is taken before the class itself. Assume we're taking one CS class per semester.

(a) The list of prerequisites for each course is given below (not necessarily accurate to actual courses!). Draw a graph to represent our scenario.

- CS 61A: None
- CS 61B: CS 61A
- CS 61C: CS 61B
- CS 70: None
- CS 170: CS 61B, CS 70
- CS 161: CS 61C, CS 70

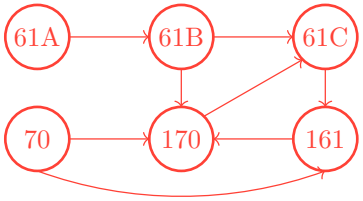
Solution:



(b) Suppose we added a new prerequisite where the student must take CS 161 before CS 170 and CS 170 before CS 61C. Is there still a valid ordering of classes such that no prerequisites are broken? If no, explain.

Solution:

The new graph looks like this:



There exists a cycle between 161 $\rightarrow$ 170 $\rightarrow$ 61C $\rightarrow$ 161, so a valid ordering does not exist. Our graph must be directed and acyclic for a topological sort to work.

(c) With the original graph, perform a topological sort to find a valid ordering of the 6 classes. Break ties by going to the lower course number first.

Solution:

With topological sorting, if an edge from vertex u to vertex v exists in the graph, then u must come before v in the sorted order. Every edge in our graph represents a prerequisite where class u must be taken before v. So, our topologically sorted order will ensure that we meet all prerequisites!

To topological sort on a graph, perform DFS from each vertex with indegree 0 (no incoming edges), but don't clear the node's we've marked between each new DFS traversal. Afterwards, we reverse the postorder to get our topologically sorted order.

In our graph, there are two vertices with indegree 0: CS 61A and CS 70. If we DFS from CS 61A then CS 70, we get a postorder of: [CS 161, CS 61C, CS 170, CS 61B, CS 61A, CS 70]. Reversing this order gives a valid ordering of: [**CS 70, CS 61A, CS 61B, CS 170, CS 61C, CS 161**].

If we DFS from CS 70 then CS 61A, we get a postorder of: [CS 161, CS 170, CS 70, CS 61C, CS 61B, CS 61A]. Reversing this order gives a different but still valid ordering of: [**CS 61A, CS 61B, CS 61C, CS 70, CS 170, CS 161**].

3 Hashing

(a) Here are five potential implementations of the `Integer` class's `hashCode()` method. Categorize each as (1) invalid, (2) valid but not good, and (3) valid and good. If it is invalid, explain why. If it is valid but not good, point out a flaw or disadvantage. For the 2nd implementation, note that `intValue()` will return that `Integer`'s number value as an `int`, and assume that `Integer`'s `equals` method checks for equality of the compared `Integer`s' `intValue`s.

```
public int hashCode() {
    return -1;
}
```

Solution: Valid but not good. As required, this hash function returns the same `hashCode` for Integers that are `equals()` to each other. However, this is a terrible hash function because collisions are extremely frequent (collisions occur 100% of the time).

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

Solution: Valid but not good. Similar to (a), this hash function returns the same `hashCode` for Integers that are `equal`. However, integers that share the same absolute values will collide (for example, `x = 5` and `x = -5` will have the same hash code). A better hash function would be to just return the `intValue` itself.

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

Solution: Invalid. When we call `super.hashCode()` here, we will ulimately end up returning `Object.hashCode()`. This hash function returns some number corresponding to the Integer object's location in memory.

However, note what happens below:

```
Integer x = new Integer(5);
Integer y = new Integer(5);
```

Here, `x` and `y` are both instantiated as separate objects, meaning they'll be at two distinct memory addresses! Then `x` and `y` would receive different hashcodes according to our current `hashCode()` implementation. However, this is not desired/valid behavior, because `x` and `y` are equal according to the `equals()` method, so their hashcodes should be the same.

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

Solution: Invalid. This function shouldn't ever have collisions (except for possible collisions as a result of effectively truncating the `long` to an `int`), but it's also not consistent because of this: calling `hashCode()` on the same object twice will result in two different `int`s.

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

Solution: Valid and good. This function will return the same `hashCode` for Integers that are `equals`, and it is consistent (always returns the same value for the same object). It is also a good hash function; there will actually be no collisions here and generally distributes elements evenly, as there is no overlap in `intValue` for two distinct integers (and by proxy, there will be no overlap in `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, "Elisa")` 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 `"Elisa"`? Explain.

   Solution: Sometimes. If the `hashCode` for the key happens to change as a result of the modification, then we won't be able to retrieve the entry in our hashtable (unless we were to recompute which bucket the new key would belong to). Changing the key can potentially (and likely) change which bucket a key would now be indexed to.

   In our example, let us suppose the `hashCode` for a key was just it's integer value, and the bucket was calculated as that number modulo the number of buckets. Let's say we had 10 buckets. In this scenario, 303 would be mapped to bucket 3, and 304 would be mapped to bucket 4. So when we `put(303, "Elisa")`, this item goes to bucket 3. Then we change the key to be 304, but don't change anything else, so this item remains in bucket 3. Now, let us suppose we later do `get(304)`. We will compute which bucket the key 304 would correspond to, which would be bucket 4. We look in bucket 4 and don't see the time there, so we cannot successfully return the item.

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, "Elisa")` and then changed that item's value from `"Elisa"` to `"Daniel"`. If we later do `get(303)`, will we be able to find and return `"Daniel"`? Explain.

   Solution: Always. The bucket index for an entry in a `HashMap` is decided by the key, not the value. Mutating the value does not affect the lookup procedure.

   In our example this would work, because when we do `get(303)` we will still go to the correct bucket.

4 Extra: 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.

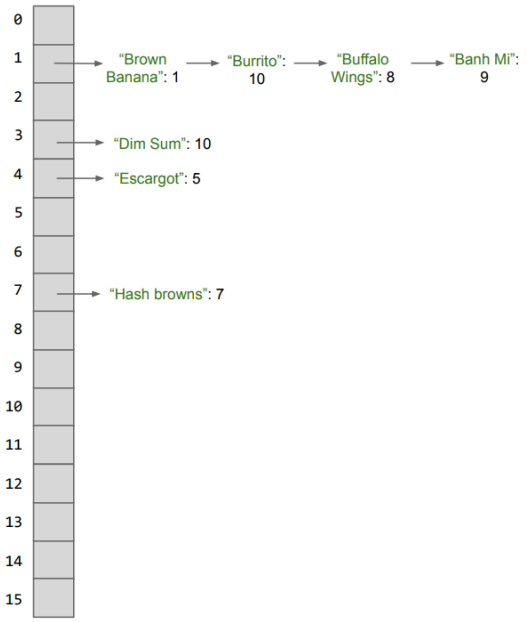
Our **`HashMap`** will compute the index as the key's hashcode value modulo the number of buckets in our **`HashMap`**. Assume the initial size is 4 buckets, and we double the size of our **`HashMap`** as soon as the load factor reaches 3/4. If we try to put in a duplicate key, simply replace the value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

Solution:

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!