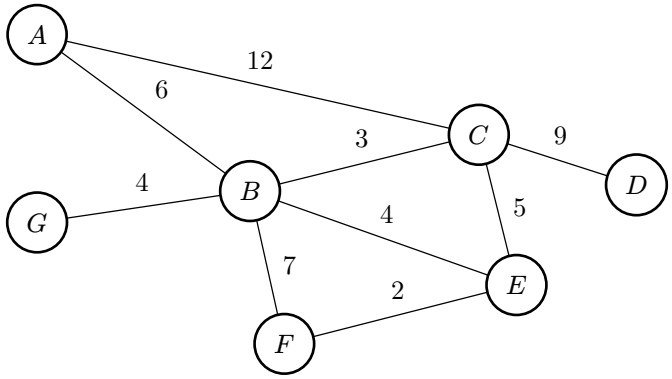
CS 61B
Fall 2025

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:

(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.

(c) True/False: If all the weights in an MST are unique, 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.

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

(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.

(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.

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;
}
```

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

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

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

```
public int hashCode() {
    return 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.

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.

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);
```

(b) Do you see a potential problem here with the behavior of our **`HashMap`**? How could we solve this?