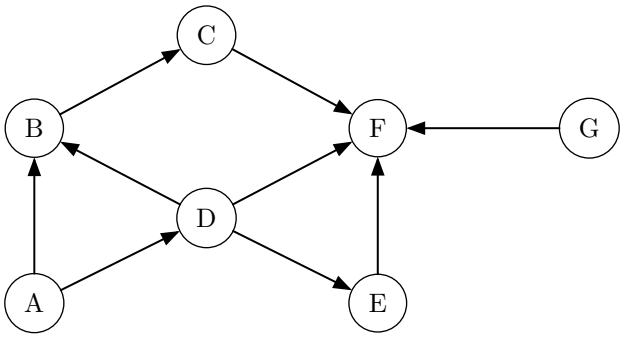
CS 61BL Summer 2026

Graphs, Hashing, and Heaps

Exam Prep 03: July 24, 2026

1 Graphs and Traversals

For the following graph…



(a) Write out the adjacency matrix and adjacency list.

Matrix:

	A	B	C	D	E	F	G
A	0	1	0	1	0	0	0
B	0	0	1	0	0	0	0
C	0	0	0	0	0	1	0
D	0	1	0	0	1	1	0
E	0	0	0	0	0	1	0
F	0	0	0	0	0	0	0
G	0	0	0	0	0	1	0

List:

```
A: {B, D}
B: {C}
C: {F}
D: {B, E, F}
E: {F}
F: {}
G: {F}
```

(b) Write out the order in which nodes are visited by DFS pre-order and post-order, starting from A. Tiebreak by alphabetical order, where nodes that come earlier in the alphabet are higher priority.

Pre-order: ABCFDE (G) Post-order: FCBEDA (G)

To compute this, we maintain a stack of nodes, and a marked set. As soon as we add something to our stack, we note it down for preorder. The top node in our stack represents the node we are currently on, and the marked set represents nodes that have been visited. After we add a node to the stack, we visit its lexicographically next unmarked child. If there is none, we pop the topmost node from the stack and note it down for postorder. Note that there are two ways DFS could run: with restart or without; DFS with restart is the version where if we have exhausted our stack, and still have unmarked nodes left, we restart on the next unmarked node.

(c) Give a valid topological sort of the graph.

`G, A, D, B, C, E, F`

…but others exist!

Answer the following questions as either **True** or **False** and provide a brief explanation.

(d) If a graph with n vertices has n − 1 edges, it must be a tree.

False. The graph must be connected as well in order for it to be considered a tree (ie. all nodes are fully connected to each other), but here some of the n - 1 edges could be edges that form a cycle instead of connecting two unconnected nodes.

(e) Every edge is looked at exactly twice in every iteration of DFS on a connected, undirected graph.

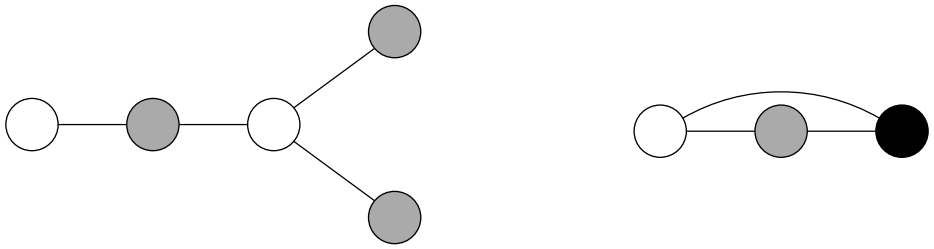
True. The two vertices the edge is connecting will look at that edge when it's their turn. Suppose that we start at u. u will traverse through u-v to reach v, then u is marked as visited. Then, once we evaluate all neighbors of v, we'll traverse through u-v one more time before noticing that u is marked as visited.

(f) In BFS, let d(v) be the minimum number of edges between a vertex v and the start vertex. For any two vertices u, v in the fringe, $|d(u) - d(v)|$ is always less than 2.

True. Suppose this was not the case (contradiction!). Then, we could have a vertex 2 edges away from the start vertex and a vertex 4 edges away in the fringe at the same time. But, the only way to have a vertex 4 edges away is if a vertex 3 edges away was removed from the fringe. We see this could never occur because the vertex 2 edges away would be removed before the vertex 3 edges away.

2 Painting Graphs

An undirected graph is said to be bipartite if each of its vertices can be painted with one of the two colors, such that every edge connects two vertices of different colors. For example below, the graph on the left is bipartite, whereas on the graph on the right is not.



Provide an algorithm which determines whether or not a graph is bipartite. What is the runtime of your algorithm?

Hint: Can you modify an algorithm we already know?

To solve this problem, we run a special version of a traversal from any vertex. This can be implemented using either DFS and BFS as the underlying traversal that we will modify. Our special version marks the start vertex as "white", then each of its neighbors "red", and each of their neighbors as "white", and so forth. If at any point in the traversal we want to mark a node as white but it is already marked with as red (or vice versa), then the graph is not bipartite.

If the graph is not connected, we repeat this process for each connected component. If the algorithm completes, successfully marking every vertex in the graph, then it is bipartite.

The runtime of the algorithm is the same whether you use BFS or DFS: $\Theta(E + V)$.

3 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.

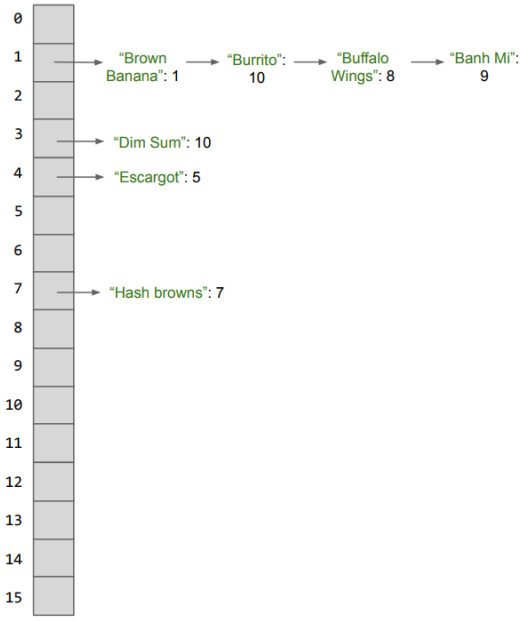
Our **`HashMap`** will compute the index as the key's hashcode value modulo the number of buckets in our **`HashMap`**. Assume the initial size is 4 buckets, and we double the size of our **`HashMap`** as soon as the load factor reaches 3/4. If we try to put in a duplicate key, simply replace the value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

4 Hashing

(a) Here are five potential implementations of the `Integer` class's `hashCode()` method. Categorize each as having (1) Good spread, (2) Bad spread, and (3) No spread.

For the 2nd implementation, note that ***`intValue()`*** *will return that* ***`Integer`****'s number value as an* ***`int`****.*

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

No spread

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

Good spread

The exact reason is somewhat out of scope of this course, but Java's `System.identityHashCode` does bit manipulation to ensure good spread.

Reasoning that memory addresses are roughly random and should, as a result, have good spread is enough.

This implementation isn't ideal for most objects, though, because it's not always valid.

```
public int hashCode() {
    return intValue() * intValue() * 2;
}
```

Bad spread

Integers that share the same absolute values will collide (for example, `x = 5` and `x = -5` will have the same hash code). Additionally, if the hash table has an even number of buckets, only half of the buckets will ever be used. A better hash function would be to just return the `intValue` itself.

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

Good spread

Note: This `hashCode` may have good spread, but it will also yield incorrect results if used with a `HashMap` or `HashSet`. This is because the `hashCode` is not deterministic. For example, if you inserted an item into a`HashSet` and it landed in bucket 4 based on the random time, then checked to see if that exact item is there, it's possible that the hashCode would be 7 the next time, causing it to look in the wrong bucket and incorrectly return false.

(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, "Dawn")` 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 `"Dawn"`? Explain.

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, "Dawn")`, 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 recpmpute the key and run `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, "Dawn")` and then changed that item's value from `"Dawn"` to `"Michelle"`. If we later do `get(303)`, will we be able to find and return `"Michelle"`? Explain.

   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.

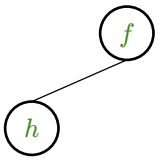
5 Absolutely Valuable Heaps

(a) Assume that we have a binary min-heap (smallest value on top) data structure called **`MinHeap`** that has properly implemented the **`insert`** and **`removeMin`** methods. Draw the heap and its corresponding array representation after each of the operations below:

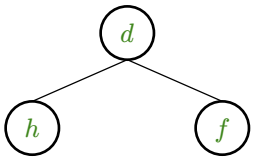
```
MinHeap<Character> h = new MinHeap<>();
h.insert('f');
h.insert('h');
h.insert('d');
h.insert('b');
h.insert('c');
h.removeMin();
h.removeMin();
```

after inserting 'f': `[-, 'f']`

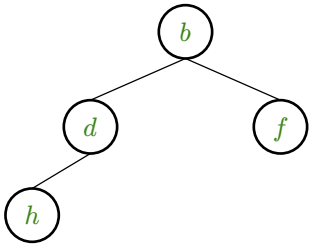
after inserting 'h': `[-, 'f', 'h']`



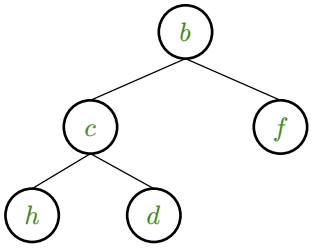
after inserting 'd': `[-, 'd', 'h', 'f']`



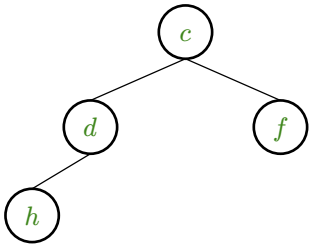
after inserting 'b': `[-, 'b', 'd', 'f', 'h']`



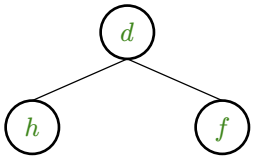
after inserting 'c': `[-, 'b', 'c', 'f', 'h', 'd']`



after removing min: `[-, 'c', 'd', 'f', 'h']`



after removing min: `[-, 'd', 'h', 'f']`



(b) Your friendly TA Benjamin challenges you to create an integer max-heap without writing a whole new data structure. Can you use your min-heap to mimic the behavior of a max-heap? Specifically, we want to be able to get the largest item in the heap in constant time, and add things to the heap in $\Theta(\log n)$ time, as a normal max heap should.

Hint: You should treat the `MinHeap` as a black box and think about how you should modify the arguments/return values of the heap functions.

Yes. For every insert operation, negate the number and add it to the min-heap.

For a `removeMax` operation call `removeMin` on the min-heap and negate the number returned. Any number negated twice is itself, and since we store the negation of numbers, the order is now reversed (what used to be the max is now the min).

Small note: There's actually one exception in Java to what we said about negation above: -2^{-31}, the most negative number that we can represent in Java, will not be itself when negated twice. This is mostly due to number representation constraints in code, but you don't need to worry about that for this question.