CS 61BL Summer 2026

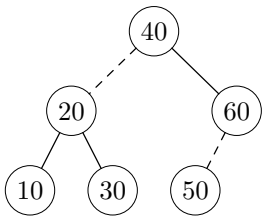
Final Exam Review

Exam Prep 05: August 7, 2026

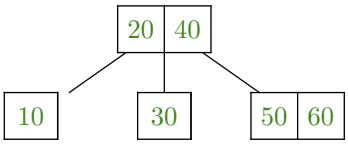
1 LLRBs

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`.

2 Bob the Shopkeeper

from Summer 2025 Final

For this question, all subparts are independent and the code compiles.

For each subpart, mark whether the given object's **`hashCode`** implementation is valid. Then, write the best case and worst case runtimes for the given scenario in the form $\Theta(\cdot)$.

You may assume that Java's **`HashSet`** implementation:

- Uses external chaining with linked list buckets
- Has an initial array capacity of 8
- Resizes by doubling the array capacity with a load factor threshold of 0.5

(a)
```
class Vi {} // intentionally empty
```

Is **`Vi`**'s **`hashCode`** valid? ● Yes ○ No

Two **`Vi`** objects are equal if and only if they share the same memory address, which is consistent with the default **`hashCode`** implementation.

(b) In total, what is the best case and worst case amortized runtime (in terms of N) of inserting N new **`Vi`** objects into an empty **`HashSet`**?

Best Case: $\Theta(N)$ Worst Case: $\Theta(N^2)$

Best case: Each object ends up in a different bucket, so it takes constant time to insert each object. There are N objects to insert, so the runtime is $\Theta(N)$.

Worst case: Each object ends up in the same bucket (even after resizing). To insert the first object will take 1 unit of time, then 2, then 3, ..., up to N. This is $1 + 2 + 3 + ... + N \in \Theta(N^2)$.

(c)
```
class Jinx {
    @Override
    public int hashCode() {
        return -982347079;
    }
}
```

Is **`Jinx`**'s **`hashCode`** valid? ● Yes ○ No

Two **`Jinx`** objects will always have the same **`hashCode`** (which is valid, since that means two equal **`Jinx`** objects will also have the same **`hashCode`**).

(d) In total, what is the best case and worst case amortized runtime (in terms of N) of inserting N new **`Jinx`** objects into an empty **`HashSet`**?

Best Case: $\Theta(N^2)$ Worst Case: $\Theta(N^2)$

Each object always ends up in the same bucket (even after resizing). To insert the first object will take 1 unit of time, then 2, then 3, …, up to N. This is $1 + 2 + 3 + ... + N \in \Theta(N^2)$.

(e)
```
class Caitlyn {
    private static int count = 0;
    public Caitlyn() {
        count += 1;
    }

    @Override
    public boolean equals(Object o) {
        quadratic(count);
        return true;
    }
}
```

Assume `quadratic(N)` runs in $\Theta(N^2)$ time.

Is `Caitlyn`'s `hashCode` valid? ○ Yes ● No

`Caitlyn` objects are always equal but can have different memory addresses (default `hashCode` implementation), which is invalid.

(f) We create N new `Caitlyn` objects. After all N `Caitlyn` objects are created, we insert each of them into an empty `HashSet`.

In total, what is the best case and worst case amortized runtime (in terms of N) for this entire process?

Best Case: $\Theta(N)$ Worst Case: $\Theta(N^3)$

Since each `Caitlyn` object is created before any insertion happens, the static variable `count` will be N by the time we insert the objects, so `equals` runs in $\Theta(N^2)$ time.

Best case: If no two objects ever end up in the same bucket (which is possible since the load factor is $0.5 < 1$), then the `equals` method is never called, so the quadratic runtime has no effect. We insert N objects with constant runtime each, so that is $\Theta(N)$.

Worst case: Every object inserted (except the first) has a collision. In this case, $N-1$ objects will call `equals` and take $\Theta(N^2)$ time each, for a total runtime of $\Theta(N^3)$. Note that since the `equals` method always returns true, there will never be more than one object in a bucket.

3 Borbders

from Fall 2025 Midterm 2

Bob loves walking between countries, which are represented by the class below:

```
public class Country {
  public String name; // Name of the country.
  public List<Country> borders; // A list of bordering countries. Is never null.
}
```

Borders go both ways. If countryA appears in countryB's borders, then countryB appears in countryA's borders.

Implement the `WalkManager` class, which has two methods:

- `public WalkManager(List<Country> countries)`:

  Constructor. Given the list of all N countries in the world, prepare any useful instance variables. Must run in $O(N^2 \log N)$ time.

- `public boolean canWalk(Country a, Country b)`:

  Returns whether country `a` is reachable from country `b`, using **one or more borders**. Must run in $O(\log N)$ time.

Here is an example in a world that has $N = 4$ countries:

- `USA` borders [`Canada`, `Mexico`].
- `Canada` borders [`USA`].
- `Mexico` borders [`USA`].
- `Ethanland` borders [] (empty list).

```
WalkManager wm = new WalkManager(List.of(USA, Canada, Mexico, Ethanland));
```

`wm.canWalk(Canada, Mexico)`; Returns `true`: `Canada` to `USA` to `Mexico`.

`wm.canWalk(Ethanland, USA)`; Returns `false`: No sequence of borders exists between `Ethanland` and `USA`.

The skeleton code begins on the next page.

Note: You may use the `WeightedQuickUnion` class from `algs4`. As a reminder, it has the following methods:

- `WeightedQuickUnion(int n)`: Constructor. Initializes a new `WeightedQuickUnion` with size `n`.
- `union(int a, int b)`: Unions the sets that `a` and `b` belong to.
- `isConnected(int a, int b)`: Checks if `a` and `b` are in the same set.

Implement WalkManager according to Bob's requirements.

```
public class WalkManager {
  // Add any instance variables below (if necessary)
  Map<Country, Integer> wquIndex;
  WeightedQuickUnion wqu;

  public WalkManager(List<Country> countries) {
    int N = countries.size();
    wquIndex = new HashMap<>();
    wqu = new WeightedQuickUnion(N);

    for (int i = 0; i < N; i += 1) {
      Country curr = countries.get(i);
      wquIndex.put(curr, i);
    }

    for (Country curr : countries) {
      for (Country other : curr.borders) {
        wqu.union(wquIndex.get(curr), wquIndex.get(other))
      }
    }
  }

  public boolean canWalk(Country a, Country b) {
    return wqu.isConnected(a, b);
  }
}
```

Graph-based solutions like `Map<Country, List<Country>> adjList` will not work, because they cannot efficiently handle cases of traversing multiple borders. For example, consider the Canada-to-USA-to-Mexico example given in the question.

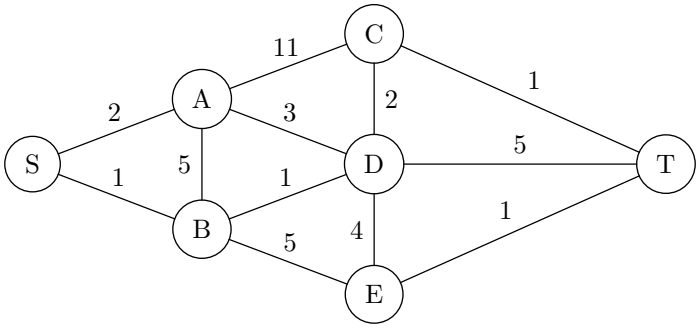
4 Lost in the Ceryneian H.I.N.D.

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.

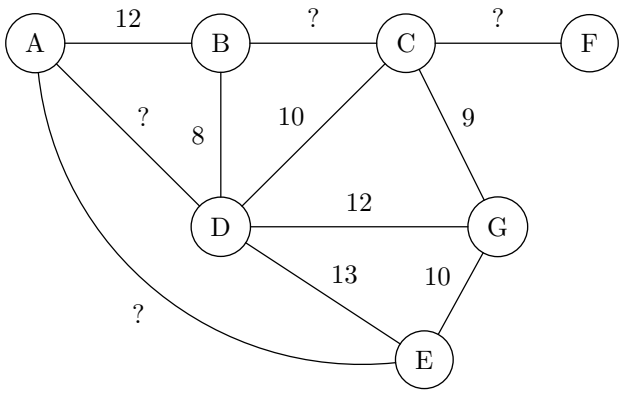
5 A Very Kapeeshable MST

from Summer 2023 Final

Omar needs to save the endangered monkfish, monkdogs, and monkbears that are scattered across the world at various locations. As a result, he creates a weighted, undirected graph G where vertices represent locations, edges represent routes, and edge weights correspond to the cost of taking a route. He wants to find an MST in G that will allow him to visit all the locations with the cheapest total cost.

However, his internet goes out before he can finish writing down all the costs of the routes!

Consider the graph G below:



(a) Choose all edges that must exist in every possible MST of G. Edges are represented as unordered sets, i.e. `{A, B}` is equivalent to `{B, A}`.

☐ `{A, B}` ☐ `{A, E}` ■ `{B, D}` ■ `{C, F}` ☐ `{D, E}` ☐ `{E, G}`

☐ `{A, D}` ☐ `{B, C}` ☐ `{C, D}` ■ `{C, G}` ☐ `{D, G}`

All based on the cut property:

`{B, D}`: smallest edge crossing the cut `{B, C, F}` ∪ `{A, D, E, G}`

`{C, F}`: only edge crossing the cut `{A, B, C, D, E, G}` ∪ `{F}`

`{C, G}`: smallest edge crossing the cut `{A, B, C, D, E, F}` ∪ `{G}`

(b) Teresa wants to create a new graph M where the number of possible cuts in M is equal to the number of edges in an MST of M. Fill in the blanks with the correct numbers (non-negative integers) to create this graph. Assume the created graph is connected, and has at least one vertex and one edge.

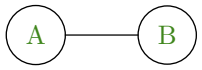
The number of vertices in the new graph must be:

2

The number of edges in the new graph must be:

1

To formalize this solution, consider the following graph G′



In G′, there exists only a single cut: {A} ∪ {B}. Since |V| = 2, the number of edges in the MST for G′ is |E| = |V | − 1 = 1. Therefore, the number of possible cuts in G′ is equal to the number of edges in its MST.

To prove that this is the only solution, observe that the number of possible cuts in any graph that is connected and has |V| > 2 is at least |V|, since you can perform a cut that partitions off each vertex from the rest of the vertices. Since an MST must only have |V| − 1 edges and the number of possible cuts is ≥ |V|, then there is no other graph that can satisfy the problem statement.

6 Frankensort

from Summer 2024 Final

Dominic is experimenting with how he can use different sorts inside of quicksort.

Quicksort:

- *Partition the array around a chosen pivot element p*
- *Quicksort the left subarray*
- *Quicksort the right subarray*

Below are versions of quicksort Dominic is considering:

Heap-Quicksort: Modify the partitioning algorithm to mimic heapsort. Choose the leftmost element to be p. Perform in-place heapsort on the array. Terminate partitioning after popping off and inserting p in heapsort.

Insertion-Quicksort: Modify the partitioning algorithm to mimic insertion sort. Choose the center element in the array to be p (center-left if length is even). Perform insertion sort on the array. Terminate partitioning after you finish swapping p as far left as it can go in insertion sort.

MSD-Quicksort: Modify the partitioning algorithm to mimic MSD radix sort. Choose the leftmost element to be p. Stably partition the array by the most significant digit, creating 3 buckets in the following order: less than, equal to, and greater than the pivot (when comparing only that digit). Partitioning is complete when the bucketing is finished. Recursively quicksort the left and right buckets by the same digit. Recursively quicksort the center bucket by the next most significant digit.

For each sequence of intermediate states of an array during quicksort, select all of the algorithms from above which could have produced the sequence. Note that these states are not necessarily the first few intermediate states, and there may be states which are skipped. Assume the first state is the array before sorting has begun. Each sequence is valid - none are impossible.

(a)
```
61 52 83 27 38 49 74 19 67
52 38 49 19 27 61 67 74 83
49 38 27 19 52 61 67 74 83
```

■ Heap-Quicksort ☐ Insertion-Quicksort ☐ MSD-Quicksort

If this was Heap-Quicksort we'd select 61 as a pivot and then run in-place heapsort. In the second state we can see that the right side of the array is in sorted order up to 61 and the left side of the array is a valid heap. In the third state we can see the next largest element, 52, get popped off the top of the heap, inserted, and the left side remains a valid heap.

If this was Insertion-Quicksort we'd select 38 as the pivot. In state 2 we see 52, to the left of 38, which is impossible if 38 is the pivot.

If this was MSD-Quicksort we would partition around 61. In the second and third state we see that the partition does not hold and elements shift around unstably. This means it cannot be MSD-Quicksort.

(b)
```
56 22 35 43 49 78 67 84 91
22 35 43 49 56 78 67 84 91
22 35 43 49 56 67 78 84 91
```

☐ Heap-Quicksort ■ Insertion-Quicksort ■ MSD-Quicksort

If this sequence represented heapsort we would select the leftmost element, 56, as the pivot and then perform in-place heapsort. The first step of in-place heapsort is bottom-up heapification. This process involves sinking elements in reverse level-order. In the second state we can see that 56 would have had to sink below many elements which are less than it, which would be impossible since we are constructing a max-heap. Therefore we know this can't be heap-quicksort.

If this was insertion sort we would pick the center element, 49, to be the pivot. We'd then insertion sort until 49 swaps into place, meaning that we can expect all of the elements up to 49's original index to be in sorted order. We can see in the second state that this happens while elements after 49's original position are unchanged. To get to the second state we can imagine choosing the center-left element in the right partition, 67, as the next pivot. This results in 67 and 78 ordering themselves correctly. Since everything matches up, we know this could be Insertion-Quicksort.

If this was MSD-Quicksort we'd choose 56 as the pivot. In the second state we can see all elements with a first digit less than 5 to the left of 56 greater than 5 to the right of 56. Furthermore, this happens stably (ex. 43 came before 49 before partitioning and that ordering is maintained after partitioning). We can see this happen again in the third state with the left subarray being partitioned around 22 and right subarray being partitioned around 67. Since everything matches up, we know this could be MSD-Quicksort.

(c) Which of the below algorithms are guaranteed to be correct?

■ Heap-Quicksort ☐ Insertion-Quicksort ■ MSD-Quicksort

Heap-Quicksort is guaranteed to sort the right subarray by the correctness of heapsort. We are also guaranteed that the largest numbers will go to the right, which means that there won't be numbers that should be in the right partition which end up in the left partition. Applying this recursively we can conclude that this algorithm will be correct.

Insertion-Quicksort may be incorrect in the event that a number less than the pivot is to the right of the pivot in the unsorted array. Partitioning will terminate before this element gets put on the left side, so it will be stuck in the right subarray. MSD-Quicksort correctly partitions the array and also correctly orders elements by the correctness of MSD radix sort. From these properties, we can conclude that this algorithm is correct.

(d) Which of the below algorithms are stable? Consider an algorithm stable based on its output, even if the output is incorrect.

☐ Heap-Quicksort ■ Insertion-Quicksort ■ MSD-Quicksort

Quicksort is stable if its partitioning scheme is stable. Heapsort is not stable. Insertion sort is stable. The problem statement states that MSD-Quicksort stably partitions.

7 Flip Flop (Extra Practice, Recommended)

Suppose we have the flip function as defined below. Assume the method unknown returns a random integer between 1 and N, exclusive, and runs in constant time. For each definition of the flop method below, give the best and worst case runtime of flip in big Theta notation as a function of N.

```
public static void flip(int N) {
  if (N <= 100) {
    return;
  }
  int stop = unknown(N);
  for (int i = 1; i < N; i++) {
    if (i == stop) {
      flop(i, N);
      return;
    }
  }
}
```

(a)
```
public static void flop(int i, int N) {
  flip(N - i);
}
```

Best case: $\Theta(N)$	Worst case: $\Theta(N)$

Consider some arbitrary value of stop that is always returned by `unknown(N)`. When `stop = x`, we do x work inside of flip (the for loop) and recursively call `flip(N - x)` through flop. This results in a total of $\frac{N}{x}$ calls before reaching our base case, and x work per call, for a total of $\Theta(N)$ work. Note that this holds for any value of x, so our best and worst case are the same.

(b)
```
public static void flop(int i, int N) {
  int minimum = Math.min(i, N - i);
  flip(minimum);
  flip(minimum);
}
```

Best case: $\Theta(1)$	Worst case: $\Theta(N \log N)$

In the best case, `unknown` always returns 1, so `stop = 1`. This hits the base case immediately, so we make 2 calls to flip then stop for $\Theta(1)$ work.

In the worst case, `stop = N / 2`. This results in flip making 2 recursive calls to itself with the argument `N / 2`. If one were to draw out the recursive tree, there would be $\log N$ levels, each one doing N amount of work. This results in a runtime of $\Theta(N \log N)$.

You might wonder why we analyze best-worst case in terms of the value of stop, even though N is passed in as an input to unknown, and we don't care about input size in best-worst case. Indeed, N might influence the value of stop, but notice that the behavior of the function unknown might influence it as well - and the latter is what we care about in asymptotic analysis.

(c)
```
public static void flop(int i, int N) {
  flip(i);
  flip(N - i);
}
```

Best case: $\Theta\ (\ N\ )$	Worst case: $\Theta\ (\ N^2\ )$

In the best case, suppose stop = 1. Then flip(N) makes recursive calls to `flip(1)` and `flip(N - 1)`, the first of which terminates immediately in the base case. `flip(N - 1)` then calls `flip(1)` and `flip(N - 2)`. The pattern is a linear recursion: constant work per call, N calls total for $\Theta(N)$ work.

In the worst case, suppose `stop = N - 1`. Note that this case is symmetrical to the best case in terms of recursive calls; however we do work proportional to N inside of flip each time because of the for loop. The overall work is $(N-1)+(N-2)+(N-3)+...+2+1=\Theta(N^2)$