CS 61B Spring 2024

Midterm 2

Thursday, March 21, 2024

Intro: Welcome to CS61B Midterm 2!

Your Name: Solutions

Your SID: _____ Location: _____

SID of Person to your Left: _____ Right: _____

Formatting:

- ◯ indicates only one circle should be filled in. ☐ indicates more than one box may be filled in. **Please fill in the shape completely.** If you change your response, **erase as completely as possible**.
- Anything you write that you ~~cross out~~ will not be graded.
- You may not use ternary operators, lambdas, streams, or multiple assignment.

Tips:

- This midterm is worth **100 points**, and there are a lot of problems on this exam. Work through the ones with which you are comfortable first. Do not get overly captivated by interesting design issues or complex corner cases you're not sure about.
- Not all information provided in a problem may be useful, and you may not need all lines.
- **We will not give credit for solutions that go over the number of provided lines or that fail to follow any restrictions given in the problem statement.**
- There may be partial credit for incomplete answers. Write as much of the solution as you can, but bear in mind that we may deduct points if your answers are much more complicated than necessary.
- Unless otherwise stated, all given code on this exam compiles. All code has been compiled and executed before printing, but in the unlikely event that we do happen to catch any bugs in the exam, we'll announce a fix.

Write the statement below in the same handwriting you will use on the rest of the exam: "I have neither given nor received help on this exam (or quiz), and have rejected any attempt to cheat. If these answers are not my own work, I may be deducted 9,876,543,210 points on the exam."

_____

_____

_____

_____

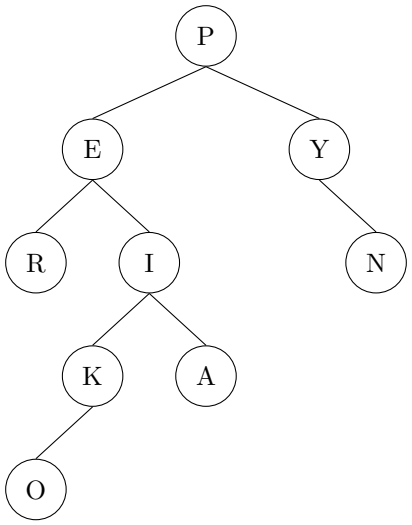
_____

Signature: _____

1 Potpourri

(22 Points)

For the first two subparts, consider the following BST of unique integer elements:



(a) How many elements must be greater than the element at node K?

5

Solution: Items I, A, P, Y, N must be greater than K, due to the search property of a BST.

(b) In what order will a post-order traversal, starting at node P, visit the nodes in the tree above?

◯ P, E, Y, R, I, N, K, A, O

● R, O, K, A, I, E, N, Y, P

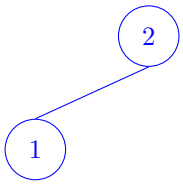
◯ O, K, A, R, I, E, N, Y, P

(c) True or false: It is possible for a rooted tree with more than 1 node to be simultaneously a valid max-heap and a valid BST.

● True

◯ False

Solution: True. An example is the following rooted tree:



(d) How many positions within a min-heap with $2^N - 1$ elements (where the height of the heap is $N - 1$) could the maximum element potentially occupy? Assume there are no duplicate elements and that the heap is not empty.

○ $\log(N)$

● $2^{(N-1)}$

○ N

○ 2^N

Solution: Since the problem states that there are no duplicate elements, we know that the maximum element is unique. Thus, it cannot be above any other element in a min-heap, and must not have any children (aka: be a leaf node).

The specification that the heap contains exactly $2^N - 1$ elements constrains the shape of the heap – namely, since our heap is binary, we know that the bottom layer of the heap must be completely full. This forces the maximum element to be on the bottom-most layer, aka depth $N - 1$ (which is also the height of our heap).

In a binary tree, the number of possible positions at any depth k is equal to 2^k. Therefore, at depth $N - 1$, there are 2^{N-1} possible positions for an element to go in.

(e) True or false: Dijkstra's algorithm always works on both directed and undirected graphs with positive edge weights.

● True

○ False

Solution: True. To run Dijkstra's algorithm on an undirected graph, you can represent each undirected edge as two directed edges (one in each direction).

(f) True or false: A* always runs asymptotically faster than Dijkstra's algorithm.

○ True

● False

Solution: False. In the worst case, A* may still need to visit all nodes, and relax all edges, before finding a solution to the goal.

(g) True or false: A* may not need to visit all nodes before completing.

● True

○ False

Solution: True. A* terminates once we reach the goal node (even if not all nodes have been visited).

(h) True or false: Kruskal's algorithm always works on graphs with negative edge weights.

● True

○ False

Solution: True. Kruskal's algorithm operates by continually adding the least-cost edge that does not form a cycle in the MST. The specific value of the edge weights do not matter, only their relative ordering.

(i) True or false: Prim's algorithm always works on graphs with negative edge weights.

● True

○ False

Solution: True. Prim's algorithm operates via the cut property, where a cut is defined as the set of edges that connect the nodes in the MST and the nodes outside of the MST. As we select the minimum weight edge in the cut for each iteration, the specific values of the edge weights do not matter.

(j) True or false: In a directed acyclic graph, any post-order traversal that visits all nodes is a valid topological sort.

◯ True

● False

Solution: False. In a directed acyclic graph, one common topological sort is **reverse** post-order traversal.

(k) In a directed tree, which of the following traversals always results in a topological sort of the tree? Assume the traversal starts at the root of the tree. Select all that apply.

■ Pre-order

☐ In-order

☐ Post-order

■ Level-order

◯ None of the above

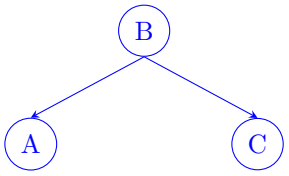
Part (k) was removed from the exam after grades were released, for the following reason: The term "directed tree" is ambiguously defined. Under reasonable assumptions, it could be interpreted to mean either:

1. A directed graph whose underlying structure is a tree, *or*
2. A directed acyclic graph with exactly one source node.

Our intended definition was to use definition 2, and that is what was assumed when providing solutions. However, due to the ambiguity, this part was cut from the exam, and scores were scaled accordingly.

Solution: Pre-order, Level-order

As an example, consider the following directed tree:



The edges tell us that B must come before A, and B must come before C, in the topological sort. Therefore, the valid topological sorts of this tree are BAC and BCA.

Now, we can consider the traversals of this tree:

- Pre-order: BAC
- In-order: ABC
- Post-order: ACB
- Level-order: BAC

In this particular example, pre-order and level-order traversals yield valid topological sorts.

To show this answer holds for all graphs, remember that in a topological sort, an edge (e.g. from node X to node Y) indicates that node X must come before node Y in the topological sort. Since we are given a directed tree in this graph, this edge also tells us that X is the parent, and Y is the child. Combining these two facts tells us that in a valid topological sort, parent nodes must always come before their children nodes.

In the pre-order and level-order traversals, the parent is visited before the children. However, in the in-order and post-order traversals, some or all of the children are visited first, before the parent is visited.

Note: Topological sort is defined for directed acyclic graphs (DAGs), but in this question, we are asked to consider topological sorts of directed trees, which are a specific subset of DAGs. In other words, all directed trees are DAGs, but not all DAGs are directed trees. For a general DAG, these traversals do not necessarily result in a valid topological sort.

2 Awesometotics

(22 Points)

For parts (a) through (d), determine the asymptotic behavior, with respect to N, of each scenario. Answer in Θ notation.

Example: A square of side length N is constructed. What is the area of that square?

$\Theta(N^2)$

(a) An `ArrayList` is implemented with additive resizing, then N elements are added to it. What is the maximum number of unused indices in the underlying array?

$\Theta(1)$

Solution: With additive resizing, when the `ArrayList` is full, you add some constant amount of space to the backing array.

Note: During the exam, many students asked clarifications about the specific implementation of the additive resizing. For example, students wanted to know how many elements were added on each resize, and whether the resize happens before or after adding a new item. These implementation details don't affect the asymptotic bound on this question. If we add a constant amount of items to the backing array, then regardless of what that exact constant is, the number of unused indices is always bounded by that constant, which does not scale with N (the number of elements added to the list).

(b) An `ArrayList` is implemented with multiplicative resizing, then N elements are added to it. What is the maximum number of unused indices in the underlying array?

$\Theta(N)$

Solution: With multiplicative resizing, when the `ArrayList` is full, the amount of space you add to the backing array is proportional to the current size of the `ArrayList`.

For example, suppose we double the size of the backing array when the backing array is full. If we add N items to the list, and the final item triggers a resize, then our backing array now has $2N$ indices, of which N are unused.

If we chose a different resizing factor, such as 5, then the backing array after a resize would have $5N$ indices, of which $4N$ are still unused.

(c) You create a weighted quick union **without** path compression, containing N elements. You then perform $\lfloor\sqrt{N}\rfloor$ calls to `union`. You look at the underlying array and take the sum of the absolute value of all elements. What is the smallest and largest possible sum?

For example, if the underlying array was $[-4, 2, 0, 0]$, the sum would be $|-4| + |2| + |0| + |0| = 6$.

Note: $\lfloor x \rfloor$ is the largest integer less than or equal to x.

Smallest sum: $\Theta(N)$

Largest sum: $\Theta(N\sqrt{N})$

Solution:
Regardless of how we call `union`, the sum of all the negative elements is a constant N, since the total weight over all trees in the WQU is exactly the number of nodes in the WQU in total. Thus we have an $\Omega(N)$ bound on the absolute value of the sum. This bound is attainable by connecting nodes 1 to $\sqrt{N}$ to node 0; regardless of tiebreaking scheme, we get $\sqrt{N}$ elements in the underlying array with value 0 or 1, which is at most a $\sqrt{N}$ total from the sum of the positive elements. Another option is to run the same union multiple times (e.g. `union(0,1)`), or to run unions that don't affect the WQU (e.g. `union(0,0)`). Thus, our best case is $\Theta(N)$.

If we connect nodes $N-2$ to $N-\sqrt{N}-2$ to node N, we would get $\sqrt{N}$ items with value $N-1$ or $N-2$ (depending on tiebreaking scheme). This yields a total among the positive elements of $\Theta N\sqrt{N}$. This is optimal; after $\sqrt{N}$ connect operations, at most $\sqrt{N}$ items in the underlying array are positive, and all items have a value at most N (the parent of a node must also be a valid node). Thus, our worst-case is $N\sqrt{N}$.

(d) A list of positive integers (not necessarily of length N) is created such that its elements sum to N^2. All elements are inserted, one at a time, into a min-heap. What are the best-case and worst-case runtimes for all insertions to complete?

Best case: $\Theta(1)$

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

Solution:
Our best case is if our list has constantly many items; for example the single-item array $[N^2]$, or the two-item array $[1, N^2-1]$. Adding constantly many items to an empty min-heap takes constant time, regardless of the size of each item. Thus, our best-case runtime is $\Theta(1)$.

The worst-case is a bit trickier. One option is to have N^2 items in the list, all of which have value 1. In this case, insertion into the heap takes constant time (since swims always fail), so our runtime would be $\Theta(N^2)$. This puts our worst-case runtime at $\Omega(N^2)$ (N^2 or slower, if there's a slower alternative). Our worst case is also $O(N^2 \log N)$, since we can have at most N^2 items, and each insertion is $O(\log N)$ time.

Our original proof that the worst-case runtime was $\mathrm{O}(N^2)$ was flawed, and have been unable to prove tighter bounds. As such, the exact proof needed to show an N^2 bound is considered out of scope; full credit was awarded for any Theta bound that is $\Omega(N^2)$ and $O(N^2 \log N)$. We suspect that $\Theta(N^2)$ is a tight bound, but please let us know if you find a better upper/lower bound!

For parts (e) and (f), give the best-case and worst-case runtimes for the functions below.

(e)
```
public static void f1(int N) {
    for (int i = 0; i < N; i += 1) {
        for (int j = i; j < N; j += 1) {
            if (j % 2 == 0) {
                i += 1;
            }
            System.out.println("naming things is hard - Dylan");
        }
    }
}
```

Best case:

○ $\Theta(1)$ ○ $\Theta(\log(\log N))$ ○ $\Theta(\log N)$ ○ $\Theta((\log N)^2)$ ● $\Theta(N)$ ○ $\Theta(N \log N)$
○ $\Theta(N^2)$ ○ $\Theta(N^2 \log N)$ ○ $\Theta(N^3)$ ○ $\Theta(N^3 \log N)$ ○ $\Theta(N^4)$ ○ $\Theta(N^4 \log N)$
○ Worse than $\Theta(N^4 \log N)$ ○ Never terminates (infinite loop) ○ None of the above

Worst case:

○ $\Theta(1)$ ○ $\Theta(\log(\log N))$ ○ $\Theta(\log N)$ ○ $\Theta((\log N)^2)$ ● $\Theta(N)$ ○ $\Theta(N \log N)$
○ $\Theta(N^2)$ ○ $\Theta(N^2 \log N)$ ○ $\Theta(N^3)$ ○ $\Theta(N^3 \log N)$ ○ $\Theta(N^4)$ ○ $\Theta(N^4 \log N)$
○ Worse than $\Theta(N^4 \log N)$ ○ Never terminates (infinite loop) ○ None of the above

Solution: Let's use the number of print statements executed as our cost model for the total amount of work done by this function. (In other words, we'll represent the print statement as doing one unit of work.)

Since this is an iterative method, let's write out the amount of work done in each iteration of the nested for loops:

- First iteration of outer for loop: We set `i=0`. In the inner for loop, we loop from `j=0` to `j=N-1`, for a total of `N` units of work.
- In the inner for loop, `j % 2 == 0` will be true `N/2` times, so this will also cause `i` to be incremented to `N/2`.
- Next iteration of the outer for loop: We set `i=N/2`. In the inner for loop, we loop from `j=N/2` to `j=N-1`, for a total of `N/2` units of work.
- In the inner for loop, `j % 2 == 0` will be true `N/4` times, so this loop will also cause `i` to be incremented by another `N/4`, to `N/2 + N/4 = 3N/4`.
- Next iteration of the outer for loop: We set `i=3N/4`. In the inner for loop, we loop from `j=3N/4` to `j=N-1`, for a total of `N/4` units of work.
- In the inner for loop, `j % 2 == 0` will be true `N/8` times, so this loop will also cause `i` to be incremented by another `N/8`, to `3N/4 + N/8 = 7N/8`.

We notice that the total amount of work done during each iteration of the outer for loop is a decreasing geometric sequence: `N` work in the first iteration, then `N/2` work in the next, then `N/4` in the next, then `N/8` in the next, etc.

Therefore, the total amount of work done by this function is `N + N/2 + N/4 + N/8 + ... + 1`, which is $\Theta(N)$.

(f)
```
public static void f2(int N) {
    if (N < 2) {
        return;
    }
    f2(N/2);
    for (int i = 0; i < Math.sqrt(N); i += 1) {
        System.out.println("Aniruth Narayanan aka the Baller");
    }
    f2(N/2);
}
```

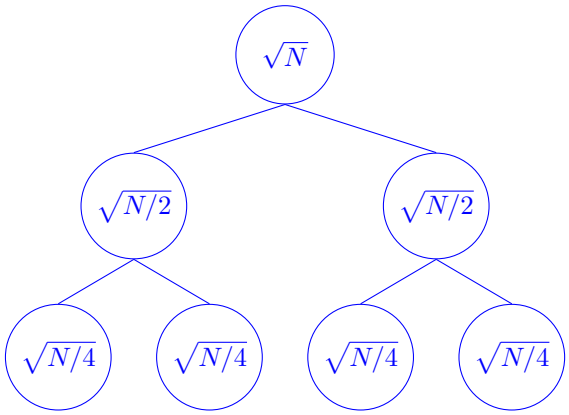
Best case:

○ $\Theta(1)$ ○ $\Theta(\log(\log N))$ ○ $\Theta(\log N)$ ○ $\Theta((\log N)^2)$ ● $\Theta(N)$ ○ $\Theta(N \log N)$
○ $\Theta(N^2)$ ○ $\Theta(N^2 \log N)$ ○ $\Theta(N^3)$ ○ $\Theta(N^3 \log N)$ ○ $\Theta(N^4)$ ○ $\Theta(N^4 \log N)$
○ Worse than $\Theta(N^4 \log N)$ ○ Never terminates (infinite loop) ○ None of the above

Worst case:

○ $\Theta(1)$ ○ $\Theta(\log(\log N))$ ○ $\Theta(\log N)$ ○ $\Theta((\log N)^2)$ ● $\Theta(N)$ ○ $\Theta(N \log N)$
○ $\Theta(N^2)$ ○ $\Theta(N^2 \log N)$ ○ $\Theta(N^3)$ ○ $\Theta(N^3 \log N)$ ○ $\Theta(N^4)$ ○ $\Theta(N^4 \log N)$
○ Worse than $\Theta(N^4 \log N)$ ○ Never terminates (infinite loop) ○ None of the above

Solution: Since this method is recursive, let's draw a tree showing the recursive calls. Each node represents a call to `f2`. The label inside each node indicates the total work done during the call represented by that node. An edge between two nodes means that the function call represented by the parent node recursively called the function call represented by the child node.



The root node represents the call to `f2(N)`. The for loop causes $\sqrt{N}$ work to be done by this call. Also, two recursive calls to `f2(N/2)` are made.

The two $\sqrt{N/2}$ nodes represent the two calls to `f2(N/2)`. The for loop causes $\sqrt{N/2}$ work to be done by each call to `f2(N/2)`. Also, each call to `f2(N/2)` makes two recursive calls to `f2(N/4)`. In total, 4 calls to `f2(N/4)` are made.

The layers of the tree represents the calls to `f2(N)`, `f2(N/2)`, `f2(N/4)`, and so on. The last layer of the tree (not pictured above) represents the calls to `f2(1)`, when we encounter the base case, where the

function returns immediately (constant amount of work), and no further recursive calls are made.

Now, we need to sum up the total amount of work done by all the recursive calls. Since the work done by each recursive call is different (e.g. the call to `f2(N)` does more work than the call to `f2(N/2)`), we cannot just count the number of nodes in the tree. We notice that at each layer of the tree, the amount of work done by all the calls at that layer are the same, so this suggests computing the total amount of work done per layer.

- Top layer, calls to `f2(N)`: $\sqrt{N}$ work in total
- Next layer, calls to `f2(N/2)`: (number of nodes) $\cdot$ (work done per node) $= 2 \cdot \sqrt{N/2}$ work in total
- Next layer, calls to `f2(N/4)`: (number of nodes) $\cdot$ (work done per node) $= 4 \cdot \sqrt{N/4}$ work in total
- Next layer, calls to `f2(N/8)`: (number of nodes) $\cdot$ (work done per node) $= 8 \cdot \sqrt{N/8}$ work in total
- Last layer, calls to `f2(1)`: (number of nodes) $\cdot$ (work done per node) $= N \cdot \sqrt{N/N}$ work in total

To work out the expression for the last layer, remember that we know the work done per node is constant (1) in the base case, which means that the denominator in the square root (representing work done per call) should double until it eventually reaches N.

Another way to find the expression for the last layer: The term inside the square root starts at N and is halved at each successive layer until the term reaches 1. In other words, the sequence (one term per layer) representing the work done per call at each layer is $\sqrt{N}, \sqrt{N/2}, \sqrt{N/4}, \ldots, 1$. This sequence has $\log(N)$ terms (the number of times we need to divide N by 2 until we reach 1). This also tells us that there are $\log(N)$ layers, since this sequence has one term per layer. Since the number of nodes doubles at each layer, we know that the kth layer of the tree must have 2^k nodes. Therefore, the last layer, which is the $\log(N)$-th layer, must have $2^{\log(N)} = N$ nodes.

Now that we have terms representing the work done at each layer, we just need to sum up all the terms to find the total amount of work done at all layers:

$$\sqrt{N} + 2\sqrt{N/2} + 4\sqrt{N/4} + 8\sqrt{N/8} + \ldots + N\sqrt{N/n}$$

We can rewrite constants as follows:

$$2\sqrt{\frac{N}{2}} = \frac{2\sqrt{N}}{\sqrt{2}} = \frac{2}{\sqrt{2}}\sqrt{N} = \sqrt{2}\sqrt{N}$$

Applying this to every term, the total amount of work can be rewritten as:

$$\sqrt{N} + \sqrt{2}\sqrt{N} + \sqrt{4}\sqrt{N} + \sqrt{8}\sqrt{N} + \ldots + \sqrt{N}\sqrt{N}$$

Now, we can factor out the $\sqrt{N}$:

$$\sqrt{N}\left(\sqrt{2} + \sqrt{4} + \sqrt{8} + \ldots + \sqrt{N}\right)$$

Note that the summation inside the parentheses is a geometric series, where every term is $\sqrt{2}$ times larger than the previous term. This sum is dominated by its largest term, which is $\sqrt{N}$. Therefore, the total amount of work simplifies to:

$$\sqrt{N}(\sqrt{N}) = \Theta(N)$$

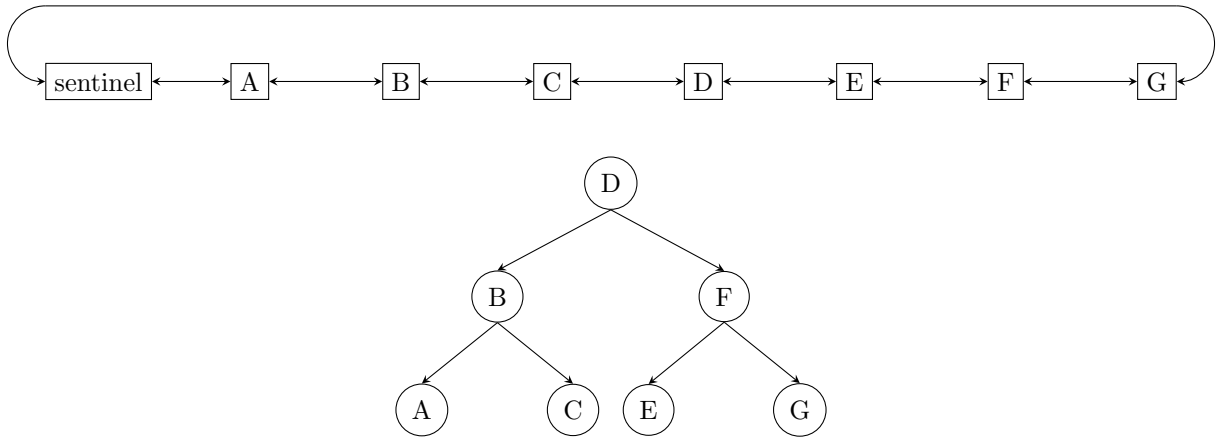
3 It's All a Blur

(18 Points)

Noah realizes that the internal nodes of DLLists and BSTs are remarkably similar: both contain an item, and two pointers to other nodes. Inspired by this, Noah creates a unified Node class that can act as the Node of either a DLList or a BST.

```
class Node {
    int item;
    Node prev;    // The previous node in a DLList, or the left child in a BST
    Node next;    // The next node in a DLList, or the right child in a BST
}
```

Noah hasn't built a wrapper class to handle DLList-specific or BST-specific operations (such as `size`). However, using these Nodes directly, Noah can connect the Nodes to create either a circular doubly-linked list with one sentinel, or a binary search tree, as in the example below.



Complete the method `treeify`, which behaves as follows:

- Input: The sentinel Node of a doubly-linked list.
- Behavior: Modifies the `next` and `prev` pointers of each Node, such that the Nodes are connected in the form of a complete binary search tree.
- Output: The root Node of the resulting tree.

In the example above, `treeify(sentinel)` should convert the top diagram to the bottom diagram, by changing `D.prev` from `C` to `B`, `D.next` from `E` to `F`, `C.next` from `D` to `null`, and so on. `treeify` should then return `D`.

You may assume:

- The input list is not empty (at least one non-sentinel Node exists).
- The number of non-sentinel items in the input list is one less than a power of two.
- Non-sentinel items in the input list are sorted in strictly increasing order.

(a) Complete `treeify` below. You may not create new Nodes. You may not use any methods of the DLList class or BST class.

```
public static Node treeify(Node sentinel) {
    return helper(sentinel.next, sentinel.prev);
}

private static Node helper(Node begin, Node end) {
    if (begin == end) {
        begin.prev = null;
        begin.next = null;
        return begin;
    }
    Node forward = begin;

    Node backward = end;
    while (forward != backward) {

        forward = forward.next;

        backward = backward.prev;
    }

    forward.prev = helper(begin, forward.prev);

    forward.next = helper(forward.next, end);

    return forward;
}
```

Solution: Recall that in a complete tree, all non-leaf nodes have two children. In other words, we want the resulting binary search tree to be as bushy as possible.

First, we need to find the node that should be the root of the tree. To keep the tree bushy, we want to place half the items to the left of the root, and half the items to the right of the root. We know from the question that the linked list has its items sorted in strictly increasing order, so the root of the tree should be the middle item. For example, in the example shown, D is the root of the tree, because it is the middle item in the linked list.

To find the middle item in the linked list, we set a pointer at the front of the linked list (`forward = begin`) and a pointer to the back of the linked list (`backward = end`). Then, we step the pointers toward each other repeatedly until they meet at the middle item (this is what the while loop is doing).

Note: The question says that the number of items in the list is one less than a power of two. This guarantees that our list has an odd number of items, and will have a single middle item (e.g. we'll never have a list of 4 items, which would have no clear middle item).

Now that we've found the root, we can recursively call the method to treeify the left subtree (all items less than the root), and the right subtree (all items greater than the root). This is what the two lines after the while loop are doing.

Note: Including `forward`/`backward` in the recursion (e.g. `helper(begin, forward)`) will not work, since the recursive call will end up with an even number of nodes, which has an ambiguous middle node.

Finally, we need to return the root node (i.e. the middle node of the original list). This also helps the recursive call work: we can set the root's `next` to be the root of the right subtree (result of calling `treeify` on the right half of the list). Similarly, we can set the root's `prev` to be the root of the left subtree (result of calling `treeify` on the left half of the list).

Alternate solutions:

Here is a non-comprehensive list of alternate solutions that would be accepted for full credit. Partial-credit answers are not listed here; we don't have that information right now, so please don't ask.

The first two lines could be `forward = begin.next`, and `backward = end.prev`. This works on length-1 lists because we enter the base case `if (begin == end)` immediately. This also works on longer lists (e.g. length-3, length-7, etc.) because the while loop just runs one fewer time.

The order of the two lines in the while loop can be switched (i.e. `backward = backward.prev` on the first line, and `forward = forward.next` on the second line).

The order of the two lines before the while loop can be switched (i.e. `forward = end` and `backward = begin`), though this would be kind of strange style since it contradicts the variable names we gave. However, if you make this switch, you must also change the while loop appropriately, so that `forward = forward.prev` and `backward = backward.next`.

Note that the code snippet below is not correct, because it scans backward from the start of the list, and scans forward from the start of the list. The while loop would terminate after a single iteration, and `forward` and `backward` would both end up pointing at the sentinel, instead of the middle node.

```
Node forward = begin;
Node backward = end;
while (forward != backward) {
   forward = forward.prev;
   backward = backward.next;
}
```

In the last three lines (after the while loop), we know that `forward == backward`, since the while loop has terminated (so its condition is now false). This means you can replace any appearance of `forward` with `backward`, and the code would still be correct.

For example, you could return `backward` instead of returning `forward`. Or, you could write `backward.prev = helper(begin, backward.prev)` instead of `forward.prev = helper(begin, forward.prev)`.

If you decide to change either `forward` or `backward` after the while loop (which the provided solution does not do), then it would probably no longer be correct to use `forward` and `backward` interchangeably.

Here's another alternate solution where we swap the definition of `forward` and `backward` in the entire code:

```
Node forward = end;
Node backward = begin;
while (forward != backward) {
   forward = forward.prev;
   backward = backward.next;
}
```

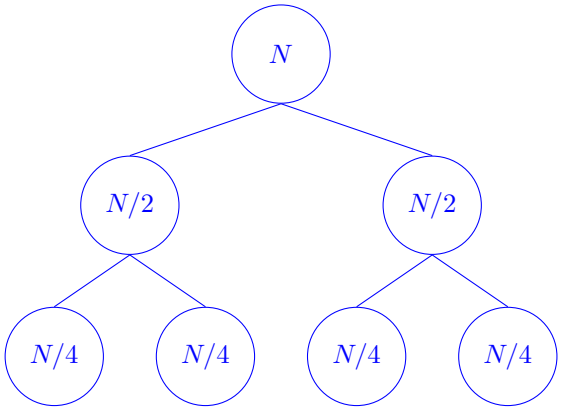
(b) What is the runtime of `treeify`, if run on a list of `N` nodes, where `N` is one less than a power of 2? Express your answer as a Θ bound.

$\Theta(N \log N)$

Solution: Excluding the recursive calls, our helper function performs $\Theta(N)$ work, where N is the size of the input list. This is because the while loop must run for $N/2$ iterations to find the middle item, and the reassigning of pointers inside the while loop takes constant-time (i.e. each iteration of the loop runs in constant time).

Now that we know how much work the function performs, we need to account for the recursive calls. Let's draw a recursive tree:

Each node represents a call to `helper`. The label inside each node indicates the total work done during the call represented by that node. An edge between two nodes means that the function call represented by the parent node recursively called the function call represented by the child node.



The root node represents the initial call to `helper` with the entire list (length `N`). The for loop causes `N` work to be done by this call. Also, two recursive calls are made, one for the left half of the list, and one for the right half of the list.

The two $N/2$ nodes represent the two calls to the left and right half of the list. The for loop causes $N/2$ work to be done by each call. Also, each call makes two more recursive calls, passing in lists of length $N/4$ (half of the $N/2$ list). In total, 4 more recursive calls are made.

A total of N work is done at each layer: N at the top layer, $2 \cdot N/2 = N$ work at the next layer, then $4 \cdot N/4 = N$ work at the next layer, and so on. (For each layer, we're multiplying the number of nodes/calls by the work done per node/call at that layer.)

The base case occurs when we pass in a list of length 1. The top layer has a list of length N, the next layer has lists of length $N/2$, the next layer has lists of length $N/4$, and so on, until we reach a layer with lists of length 1. There are $\log(N)$ layers, because that is how many times we need to halve N before reaching 1.

Since there are $\log(N)$ layers and each layer performs $\Theta(N)$ work, our total runtime is $\Theta(N \log(N))$

Alternate solution (compare to an algorithm we've seen before): This function does a linear amount of work (to find the middle item), then makes two recursive calls on the left and right halves. This is exactly what mergesort does, and mergesort runs in $\Theta(N \log N)$ runtime.

Disclaimer: The strategy of comparing to other algorithms only works if you're very confident that the function behaves exactly the same as another algorithm you've seen before. Sometimes, two algorithms look very similar, but end up having different runtimes, so use this approach with caution on other questions.

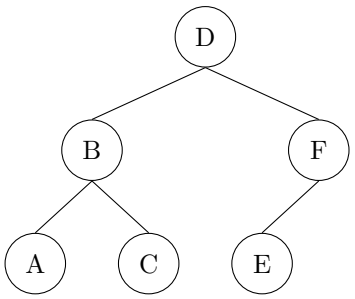
4 Lost in LLRBs

(18 Points)

After finishing your Deque from Midterm 1, you start working on an LLRB implementation. However, the same hacker comes along and paints all your links black! You may assume that the LLRB was valid before it got hacked.

Recall from lab that an LLRB node is red if the link to its parent is red, and black if either the node is the root, or the link to its parent is black. For example, if the edge E-F is meant to be red, then E is colored red.

(a) Consider the following hacked LLRB (all links painted black):



Select all nodes that must be red for this LLRB to be valid.

☐ A ☐ C ☐ F

■ B ■ E ◯ None of the above

(b) Write `fixLLRB`, which restores the correct colors to all nodes without otherwise modifying the tree.

Your code must be asymptotically optimal. You may not use any LLRB methods/attributes not listed below.

```
public class LLRB {
    private class Node {
        int value;
        Node left;
        Node right;
        boolean isRed;
    }

    private Node root;

    public void fixLLRB() {
        helper(root);
    }

    private int helper(Node n) {

        if (n == null) {

            return 0;
        }
        int leftHeight = helper(n.left);

        int rightHeight = helper(n.right);

        if (leftHeight != rightHeight) {
            n.left.isRed = true;
        }

        return rightHeight + 1;
    }
}
```

Solution:

The main crux of this problem is to determine an identifying aspect of a red link. The simplest aspect is that a red link does not decrease height in the corresponding B-tree. Therefore, we let the return value of the helper function be the height of the node in the corresponding B-tree. Since the right edge is always black, we can guarantee that the height of the right subtree is one less than the height of the current node. We can then recursively check if the left subtree has that same height as the right subtree (in which case the left link is black) or one more than the right height (in which case the left link is red), and return one more than the right height (which is guaranteed to be the height of the corresponding B-tree rooted at the current node).

Alternate solutions/Grading rubric:

In an attempt to preemptively reduce questions in future semesters, we have provided the grading rubric below, along with augmented explanations.

Your answer must EXACTLY match one of our descriptions in order to receive a rubric item. **There is no partial credit for answers that are similar but do not exactly match our rubric items.**

Blank 1:

This was graded all-or-nothing, with `int` being the only accepted answer.

Blank 2:

`n == null` is the only full-credit solution.

Partial credit was given for the following answers, which work for most cases but fail on null (empty) LLRBs:

- `n.left == null`
- `n.left == null && n.right == null`

Blank 3:

This was graded all-or-nothing, with any constant number (e.g. 0, or 1, or 2, etc.) being accepted. This causes `helper` to return the height of the tree, plus some constant. However, we're still able to compare the height of the left and right subtrees, since the same constant was added to each subtree height.

Blank 4:

To get this rubric item, blank 4 must be EXACTLY `helper(n.left)` (plus or minus a constant). Examples include:

- `helper(n.left) + 1` can get this rubric item (matches the answer, plus a constant)
- `helper(n.left)` can get this rubric item
- `helper(n.left.value)` does NOT get this rubric item, because it does not match the answer exactly
- Answers that contain `helper(n.left)`, but also contain other code, do NOT get this rubric item, unless they are EXACTLY `helper(n.left) + 1` (possibly replacing `1` with another constant value)

—

This rubric item was also applied if ALL of these are true: You received points for Blank 6, AND your answer for Blank 6 does not use `leftHeight`, AND and your answer for Blank 4 runs in constant time. This is to account for answers where Blank 6 directly calls `helper(n.left)`, and Blank 4 is therefore unnecessary. For example:

- Blank 4 answer: `int leftHeight = /* anything constant-time */`
- Blank 6 answer: `if (helper(n.left) != helper(n.right))`

Blank 5:

To get this rubric item, Blank 5 must be EXACTLY `helper(n.right)` (plus or minus a constant)

To get this rubric item, `helper(n.right)` can NOT be called a second time in Blanks 6 or 7. (This would cause the runtime of the function to not be optimal.)

Blank 6:

To get this rubric item, Blank 6 must be EXACTLY:
`leftHeight == rightHeight + 1`
OR
`leftHeight > rightHeight`
OR
`leftHeight != rightHeight`

Exactly equivalent expressions also get this rubric item.

Examples:

- `rightHeight < leftHeight` gets this rubric item (exactly equivalent, just with swapped order)
- `leftHeight - 1 == rightHeight` gets this rubric item (exactly equivalent)
- `leftHeight >= rightHeight` does NOT get this rubric item (adding the equality causes extra incorrect red links to be added)

—

This rubric item was also applied if your answer evaluates to EXACTLY:
`helper(n.left) == helper(n.right) + 1`
OR
`helper(n.left) > helper(n.right)`
OR
`helper(n.left) != helper(n.right)`

Examples:

- You wrote `leftHeight = helper(n.left)` and `rightHeight = helper(n.right) + 1`. If you defined the variables this way, then `leftHeight == rightHeight` would get this rubric item, since it evaluates to `helper(n.left) = helper(n.right) + 1`. However, if you defined the variables any other way in Blanks 4/5, then `leftHeight == rightHeight` would NOT get this rubric item.

—

Note: `leftHeight` and `rightHeight` could be defined differently, as long as later blanks are consistent.

For example, you could define `rightHeight = helper(n.right) + 1` (adding one), and return `rightHeight` (the adding one was accounted for when the variable was defined, instead of in the return statement).

As another example, you could also write:

```
1 int leftHeight = helper(n.left) + 1;
2 int rightHeight = helper(n.right) + 1;
3 if (leftHeight != rightHeight) {
4     n.left.isRed = true;
5 }
6 return rightHeight;
```

Since we added 1 to both heights, the comparison of the two heights is still the same. We can still check if the heights are not equal, or if `leftHeight > rightHeight`, or if `leftHeight == rightHeight + 1`.

—

Note: Checking `!n.isRed` is ok if ANDed to an otherwise-correct solution. For example, `leftHeight != rightHeight && !n.isRed` can get this rubric item. Adding it does not change the code behavior because the question says the LLRB was valid before it got hacked, and `n.isRed` and `n.left.isRed` would imply there are two consecutive red links, which makes the LLRB invalid. Therefore, this extra condition always returns true.

Blank 7:

To get this rubric item, Blank 7 must be EXACTLY:
`rightHeight + 1`
OR
`Math.min(leftHeight, rightHeight) + 1`

The 2nd answer works because `rightHeight` is always equal to `leftHeight`, or one less than `leftHeight`. Therefore, the minimum will always be `rightHeight`.

—

This rubric item was also applied if your answer evaluates to EXACTLY: `helper(n.right) + 1`

Examples:

- You wrote `rightHeight = helper(n.right) + 1`. If you defined `rightHeight` this way, then **`return`** `rightHeight` would get this rubric item. However, if you defined `rightHeight` any other way in Blank 5, then **`return`** `rightHeight` would NOT get this rubric item.

Miscellaneous deductions for blanks 4-7:

A small amount of points were deducted if your code contained any of the following errors:

- `height` instead of `helper`
- `n.leftHeight` instead of `leftHeight`
- `helper(left)` instead of `helper(n.left)`
- `helper(right)` instead of `helper(n.right)`
- `left.height` instead of `leftHeight`
- `right.height` instead of `rightHeight`
- `=!` instead of `!=`
- `new` where it is unnecessary, e.g. `new helper(n.left)`

5 I Love Hashbrowns

(12 Points)

In this question, consider these two `hashCodes`:

- The **default** `hashCode`: The implementation of `hashCode` provided by the `Object` class.
- The **trivial** `hashCode`: The `hashCode` that always returns 0.

Also, consider these two `equals` methods:

- The **default** `equals` method: The implementation of `equals` provided by the `Object` class.
- The **trivial** `equals` method: The `equals` method that always returns `true`.

Select whether each statement is true or false.

Recall from lecture that the default `hashCode` returns the memory address of an object.

Also, recall from lecture that the default `equals` method uses `==` to compare two objects. This method returns true if the two objects have the same memory address, and false otherwise.

Also, here is the only clarification that was issued during the exam (it didn't change any of answers, but many students asked):

5c and 5d: "all equals methods" refers to any possible equals operator, not just the two defined in the question.

(a) The **default** `hashCode` is a valid `hashCode` for the **trivial** `equals` method.

◯ True

● False

Solution: False. The trivial `equals` method considers all objects to be equal to each other. A valid `hashCode` must return the same `hashCode` for two equal objects. Therefore, for the trivial `equals` method, we need the corresponding `hashCode` to return the same value for all objects.

However, the default `hashCode` will not return the same `hashCode` for all objects.

For example: We have two different Cat objects, `Cat fish = new Cat(...)`, and `Cat chips = new Cat(...)`. The trivial `equals` method considers all objects equal, so these two Cats are equal to each other. However, the two Cats would have different `hashCodes` according to the default `hashCode` method, because they're stored in different memory locations.

(b) The **trivial** `hashCode` is a valid `hashCode` for the **trivial** `equals` method.

● True

◯ False

Solution: True. From the previous part, we know that for the trivial `equals` method, we need the corresponding `hashCode` to return the same value for all objects.

The trivial `hashCode` does return the same value for all objects, so it would be a valid `hashCode` in this case.

In other words: All equal objects (every object ever) will have the same `hashCode` (because every hash code ever is 0).

(c) The **trivial** `hashCode` is a valid `hashCode` for **all** `equals` methods.

● True

○ False

Solution: True. Let's check the three properties of valid `hashCodes`:

1. The `hashCode` returns an integer. The trivial `hashCode` does return an integer (0).

2. Calling `hashCode` twice on the same unchanged object should yield the same integer. The trivial `hashCode` always returns 0, so it will always yield the same integer.

3. Two equal objects should have the same `hashCode`. Since every `hashCode` is 0, two equal objects will both have `hashCodes` of 0, and therefore have equal `hashCodes`.

This reasoning holds for any `equals` method we choose to use, because any two objects will always have equal `hashCodes`. Therefore, any two equal objects will also always have equal `hashCodes`.

(d) The **default** `hashCode` is a valid `hashCode` for **all** `equals` methods.

○ True

● False

Solution: False. Consider the trivial `equals` method, which considers all objects to be equal. From part (a), we know that the default `hashCode` is not valid for the trivial `equals` method. Therefore, the default `hashCode` is not a valid `hashCode` for all `equals` methods (because we found a counterexample `equals` method).

(e) The **only** valid `hashCode` (over all possible `hashCodes`) for the **default** `equals` method is the **default** `hashCode`.

○ True

● False

Solution: False. Consider the trivial `hashCode`. From part (e), we know the trivial `hashCode` is a valid `hashCode` for all `equals` methods, including the default `equals` method.

We found another valid `hashCode` for the default `equals` method, so the statement is false.

(f) The **only** valid `hashCode` (over all possible `hashCodes`) for the **trivial** `equals` method is the **trivial** `hashCode`.

○ True

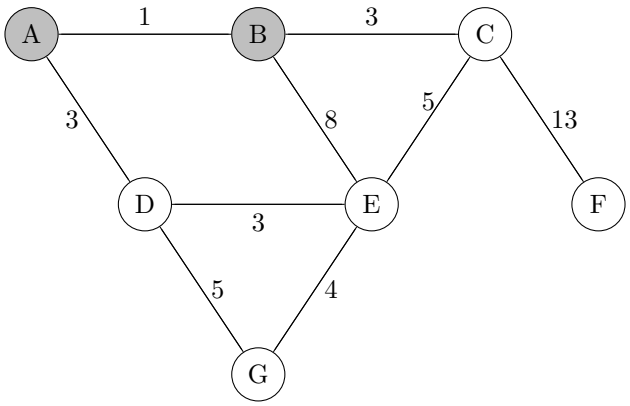
● False

Solution: False. Consider a `hashCode` that always returns 1, or a `hashCode` that always returns 5, or any other `hashCode` that always returns a constant number. All of these `hashCodes` would be valid for the trivial `equals` method.

We found another valid `hashCode` for the trivial `equals` method, so the statement is false.

6 Where's Prim?

(8 Points)



(a) Suppose we run Dijkstra's algorithm on the graph above, with source node A. After visiting A and B, the `distTo` array is as follows:

Node	A	B	C	D	E	F	G
`distTo`	0	1	4	3	9	∞	∞

Fill in the `distTo` array after visiting one more node and relaxing all of its outgoing edges. In each box, write either a number or ∞.

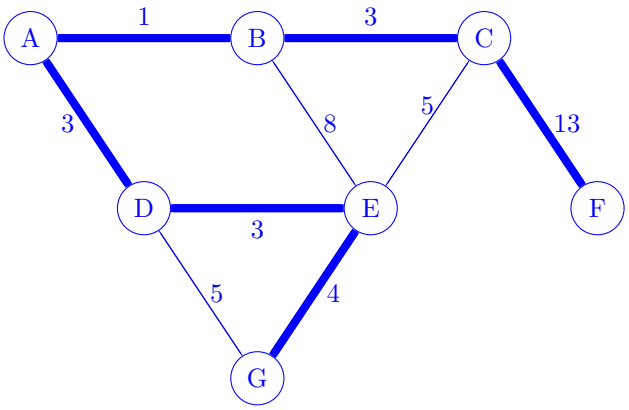
Node	A	B	C	D	E	F	G
`distTo`	0	1	4	3	6	∞	8

Solution: The next node that we will visit is node D, since it is the node with the lowest `distTo` value that we have not visited yet. After visiting D, we relax the cost of the path from 8 to E to 6 (A $\rightarrow$ D $\rightarrow$ E). We also relax the cost of the path to G from ∞ to 8 (A $\rightarrow$ D $\rightarrow$ G). The rest of the `distTo` values do not change.

(b) What is the sum of the edge weights in the minimum spanning tree of the graph above?

27

Solution: The minimum spanning tree will have a weight of 1 (AB) + 3 (AD) + 3 (BC) + 3 (DE) + 4 (EG) + 13 (CF) = 27.



(c) Which edge(s) are guaranteed to be in all (not necessarily minimum) spanning trees? Select all that apply.

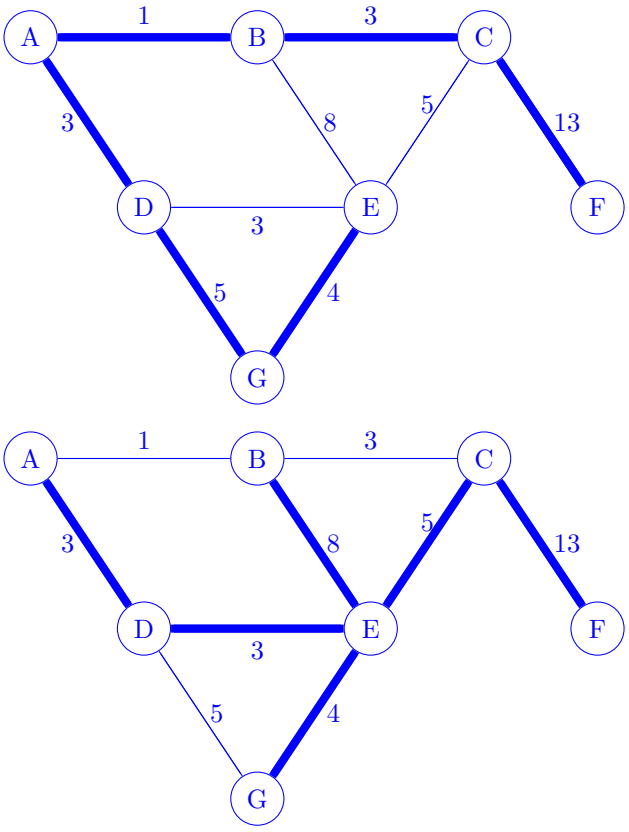
☐ AB	☐ CE	☐ EG
☐ AD	■ CF	◯ None of the above
☐ BC	☐ DE	
☐ BE	☐ DG	

Solution: CF only

Remember that a spanning tree (not a minimum spanning tree) is any set of edges that form a tree and touch all the nodes. The MST is the spanning tree with minimum weight, but that doesn't stop us from building spanning trees with non-minimal weight.

Solution 1 (draw examples):

Here are some examples of spanning trees (many others exist):



From these two examples, we can already eliminate every edge except CF. For example, AB is eliminated because the second spanning tree does not include AB, and BE is eliminated because the first spanning tree does not include BE.

(Even if you didn't see these two exact spanning trees, you could draw a few more, and would eventually be able to rule out all edges except CF by drawing a spanning edge that doesn't use that edge.)

Solution 2 (use definition of spanning tree):

We know that every node must be connected to the spanning tree by at least one edge. To connect node A to the tree, we could either use AB or AD, so we could choose to use one of the edges without using the other. This eliminates AB and AD as options. (Note that we don't necessarily need to use the edge with minimum weight, since we are constructing any spanning tree, not necessarily the minimum spanning tree.) Similarly, there are multiple choices of edges to use to reach nodes B, C, D, E, and G.

However, there is only a single choice of node to connect F to the tree. Every spanning tree, regardless of total weight, must use the CF edge.

Solution 3 (change edge weights):

Since we want any spanning tree (not necessarily the minimum spanning tree), we don't care about the edge weights. To reflect this, we can arbitrarily set every edge weight to 1. Then, we can try to find the MST of the resulting graph (where edge weights are irrelevant). Every MST of this graph will be a spanning tree of the original graph (since the graphs have the same structure).

Using the cut property, we can create a cut with only F on one side, and all other nodes on the other side. We see that CF is the only choice of edge crossing this cut, so it must be in the MST.

However, for all other cuts where one side is a single node (e.g. only A/B/C/D/E/G on one side, all other nodes on the other side), there are multiple choices of the edge crossing the cut, so none of these edges are guaranteed to be in the spanning tree.

(d) You add a new edge between F and G with weight x, and notice that the MST of the modified graph has a lower total weight (compared to the original MST). What is the maximum integer value that x could be?

12

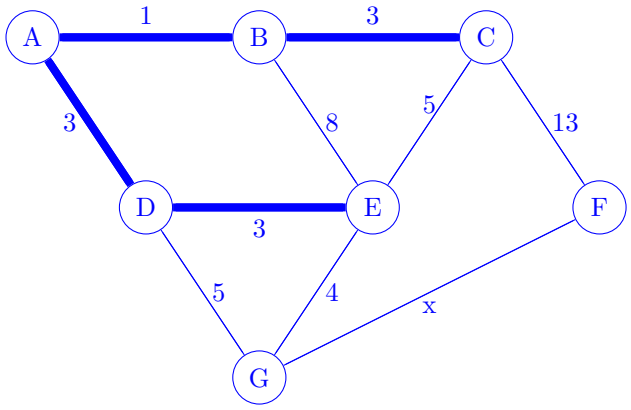
Solution:

Solution 1 (use your answer from part b):

Assuming the newly created edge FG is part of the MST, the new minimum spanning tree will have a weight of 1 (AB) $+3$ (AD) $+3$ (BC) $+3$ (DE) $+4$ (EG) $+x$ (FG) $= 14 + x$. Since we want the new MST to have a lower total weight than the original MST, we solve the inequality $14 + x < 27$, which yields $x < 13$. Since we ask for the maximum integer that x could be, our answer is 12.

Solution 2 (disconnect edges, rebuild the MST):

Since the newly added edge connects nodes F and G, we disconnect the edges of the original MST that are adjacent to nodes F and G, resulting in an intermediate MST that looks like this:



From here, we run an MST algorithm to add the last 2 edges in order to reach $V-1=6$ edges, where V is the number of vertices. We will use Prim's algorithm in the solutions, but any other MST algorithm will work too.

Using Prim's algorithm, we find that our cut contains edges {CF, DG, EG}. Of those, we add edge EG to the MST since it has the lowest weight. On the next iteration, we find that our cut contains edges {CF, FG}. Since we want to include the newly-added edge FG, FG must have a lesser weight x than all other edges in the cut. As such, we get that $x < 13$, and since we specifiy that x is an integer, $x = 12$.

Nothing on this page is worth any points.

---

7 The Dancing Men

(0 Points)

The Dancing Men Cipher was initially conceived by Sir Arthur Conan Doyle in the Sherlock Holmes story "The Adventure of the Dancing Men". The original alphabet was incomplete and only contained 18 letters, but today there is a consensus on all 26 English letters and all 10 digits.

What does the following ciphertext decode to?

J MONEY YOK

8 Feedback

(0 Points)

Leave any feedback, comments, concerns, or more drawings below!

Have a nice spring break!