

UC Berkeley – Computer Science
CS61B: Data Structures

Midterm #2, Spring 2021

This test has 8 questions across 14 pages worth a total of 1920 points, and is to be completed in 110 minutes. The exam is closed book, except that you are allowed to use unlimited written cheat sheet (front and back). No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided. **Write the statement out below in the blank provided and sign your name. You may do this before the exam begins.**

“I have neither given nor received any assistance in the taking of this exam.”

Signature: _____

#	Points	#	Points
1	205	5	370
2	120	6	200
3	75	7	350
4	195	8	405
		TOTAL	1920

Name: _____

SID: _____

GitHub Account # : sp21-s_____

Tips:

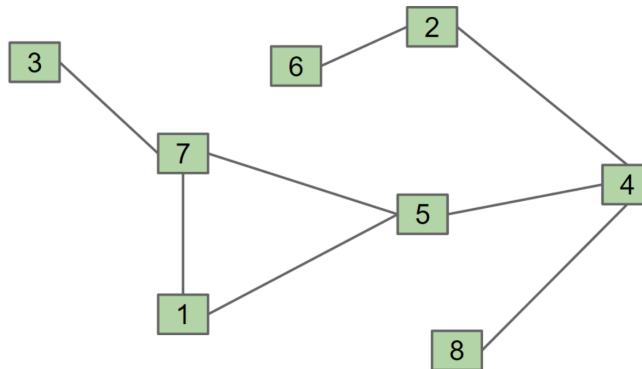
- There may be partial credit for incomplete answers. Write as much of the solution as you can, but bear in mind that we may deduct points if your answers are much more complicated than necessary.
- There are a lot of problems on this exam. **Work through the ones you are comfortable with first. Do not 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.**
- Unless otherwise stated, all given code on this exam should compile. 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. **Unless we specifically give you the option, the correct answer is not 'does not compile.'**
- indicates that only one circle should be filled in.
- indicates that more than one box may be filled in.
- For answers which involve filling in a or , please fill in the shape completely.

Optional. Mark along the line to show your feelings
on the spectrum between ☹ and ☺.

Before exam: [☹_____☺].
After exam: [☹_____☺].

1. Graph Traversals (205 points).

Consider the graph below:



a) (30 points). If we perform a BFS traversal starting from vertex 7, what will be the **final** vertex visited?

- 1 2 3 4 5 6 7 8

b) (40 points). What start vertex (or vertices) will result in a DFS pre-order of 84265173? **Assume ties are broken numerically.** If no start vertex would yield 84265173 from a pre-order traversal, select Impossible.

- 1 2 3 4 5 6 7 8 Impossible

c) (60 points) What start vertex (or vertices) will result in a DFS post-order of 84265173? **Assume ties are broken numerically.** If no start vertex would yield 84265173 from a post-order traversal, select Impossible.

- 1 2 3 4 5 6 7 8 Impossible

d) (75 points) Suppose we want a DFS pre-order of 12345678, i.e. in numerical order. In the original graph, there is no vertex that could possibly yield this order if we perform a pre-order traversal.

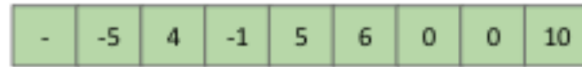
What is the minimum number of edges we'd need to add to the graph so that a pre-order traversal could yield this order from some start vertex?

- 1 2 3 4 5 6 7 8 Impossible

2. Heaps and Tree Traversals (120 Points).

Suppose we have an `ArrayHeap<Integer>` that uses the array representation from class and discussion (also called "tree representation 3B" in lecture). Recall that in this representation that the leftmost item is unused.

Consider a heap with the following underlying array:



a) **(35 points)** Suppose we perform a pre-order traversal of the heap represented by this array. What will be the **last** value in the pre-order traversal?

- 5
- 1
- 3
- 0
- 4
- 5
- 6
- 10

b) **(35 points)** Suppose we perform an in-order traversal of the heap represented by this array. What will be the **last** value in the in-order traversal?

- 5
- 1
- 3
- 0
- 4
- 5
- 6
- 10

c) **(50 points)** Suppose we are using our heap to represent a priority queue. If we call `removeMin` on the heap above, where will the 10 end up after `removeMin` has completed execution?

Assume `removeMin` works as described in lecture and discussion. By "completed execution", we mean the entire operation is done and the array again obeys our heap properties.

- 10 will not move.
- 10 will not be present in the heap.
- In the root position previously occupied by a -5.
- In a position previously occupied by a 4.
- In a position previously occupied by a 5.
- In a position previously occupied by a 6.
- In a position previously occupied by a -1.
- In a position previously occupied by a 0.

3. Space Git (75 Points)

Answer the following multiple choice questions regarding the Git version control system. For each question, choose the best answer.

a) (25 Points) Sklurp the space horse was stuck on project 1 after spending a few days on it, so Sklurp decided to start over from the skeleton code. Sklurp first deleted the proj1 directory in their sp21-s*** repo, but then realized they weren't sure what to do next to get the skeleton code to come back. Which of the following commands could Sklurp enter next to restore the working directory so that it contains the skeleton code for proj1? Assume the CWD is Sklurp's sp21-s*** repo.

- `git pull skeleton master`
- `git add *` then `git commit -m "starting over"`
- `git pull skeleton origin`
- `git push`
- `git push origin master`
- `git checkout skeleton/master -- proj1`

b) (25 Points) Later in the semester, Sklurp tried to start back over on Gitlet. After executing some other Git commands they don't remember, `git status` is telling them that their sp21-s*** repo is in a "detached HEAD" state. What does this mean? Note that this question is independent of part a).

- Their repo isn't in a clean state. Some changes need to be committed.
- They are no longer synced with the skeleton repo.
- Some files were deleted after they were added, but before they were committed.
- They tried to `git push origin master` and something went wrong, so HEAD changed its value.
- HEAD is pointing to a commit that is not pointed to by a branch.
- HEAD is pointing to corrupt files in the staging area.

c) (25 Points) What command below might possibly work to get Sklurp out of "detached HEAD" state? Below, [some hash] means some hash id, e.g. Sklurp would actually type something like `8bc2dc48260553244d2f72cf56d916a56eea96c2`, and not literally [some hash]. Again, assume the CWD is Sklurp's sp21-s*** repo.

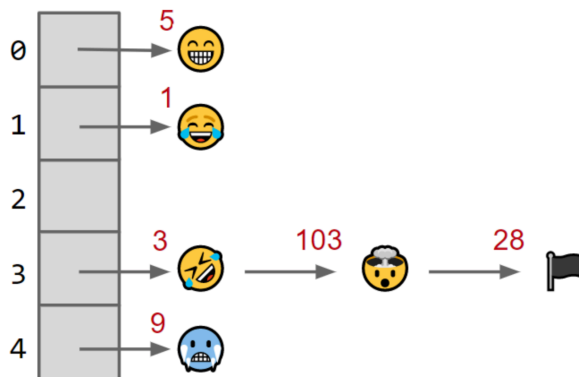
- `git checkout master`
- `git add *` then `git commit -m "starting over"`
- `git pull skeleton master`
- `git checkout [some hash] .`
- `git pull origin master`

4. Hashing. (195 points)

a) Those are the facts (40 Points). Throughout this problem, assume we're using a hashtable (as seen in lecture) to represent a set. Suppose that each bucket of the hashtable is stored as a left leaning red black tree, and we are inserting items that implement the Comparable interface. Which of the following statements are true about such a hashtable?

1) (10 points) The runtime of contains is $O(N)$.	<input type="radio"/> True <input type="radio"/> False
2) (10 points) The runtime of contains is $O(\log N)$.	<input type="radio"/> True <input type="radio"/> False
3) (10 points) The runtime of contains is $O(1)$.	<input type="radio"/> True <input type="radio"/> False
4) (15 points) One advantage of using an LLRB for the buckets is that it makes it possible to efficiently iterate over all of the keys in the set in ascending order.	<input type="radio"/> True <input type="radio"/> False
5) (15 points) Assuming items are nicely spread out in the hash table, we expect that an LLRB bucket would yield significantly better performance for contains and add than if we used an ArrayList for each bucket.	<input type="radio"/> True <input type="radio"/> False


b) Adding (120 points). Suppose now that we build a `Set<Picture>` using a hashtable, where we represent each bucket with a linked list. Suppose we've added the Picture objects below with the given hashcodes in red:



1) (40 points) We add a new picture  with hashcode -6. In which bucket will  end up in the hashtable? Assume that the hashtable does not resize.


- 0 1 2 3 4


2) (40 points) If we resize our hashtable by doubling its size, items with which hashcode will end up in a **different** bucket number than before the resize operation? Assume we're starting from the original

figure above without the  .

- 5 1 3 103 28 9

3) (40 points) Suppose that we perform the following actions on the original hashtable with 5 buckets illustrated in the image above:

1. Picture $x =$ 
2. `hashTable.add(x);` // as above, x 's hashCode is 9!
3. `x.turnPink();` // modifies x
4. `System.out.println(hashTable.contains(x));`

Assume that the `turnPink` method changes some of x 's pixels pink and adds a 3rd eye so that it looks like  . This change to the object may result in its hashCode changing.

For which of the following hashcodes will line 4 of the above code print out `true`? Note that a `Picture` object's `equals` method returns `true` only if their pixel values are exactly identical.

- `x.hashCode()` is -1 `x.hashCode()` is 0 `x.hashCode()` is 4 `x.hashCode()` is 14
- None of these

5. By the Numbers (370 Points).

a) Hash tables. For the three questions below, suppose we have an initially empty hash table with 32 buckets and which does not resize in any of these problems. The hash table starts empty for each sub-part, for example any additions done in part 1 don't affect part 2. **Do not assume anything about how the items are spread out.**

1) (20 points) If we insert 16 items into the hash table, what is the minimum number of items in bucket 0?	Answer: 0
2) (20 points) If we insert 16 items into the hash table, what is the maximum number of items in bucket 0?	Answer: 16
3) (50 points) If we insert 4 strings into the hash table, and each bucket is stored as a binary search tree (with no special balancing features), how many total string comparison operations will occur in to complete these insertions the worst case?	Answer: 6

b) BTrees and Friends. For this problem, remember that the height of a tree is the **number of edges between the root and farthest leaf.**

1) (40 points) Suppose we have a 2-3 tree of height 6. What is the minimum height (number of edges from the root to the farthest leaf) of the corresponding LLRB?	Answer: 6
2) (40 points) Suppose we have a 2-3 tree of height 6. What is the maximum height (number of edges from the root to the farthest leaf) of the corresponding LLRB?	Answer: 13
3) (60 points) What is the minimum possible height (number of edges from the root to the farthest leaf) of a 2-3 tree with 8 values?	Answer: 1
4) (60 points) What is the maximum possible height (number of edges from the root to the farthest leaf) of a 2-3 tree with 8 values?	Answer: 2
5) (80 points) What is the maximum number of values we can insert in to the 2-3 tree created in (4) without increasing its height? Warning: This problem is quite challenging.	Answer: 18

6. Disjoint Sets Iterator (200 Points).

Suppose we want to implement an iterator for `WeightedQuickUnionDS` objects. This iterator will iterate over all items in a given set in numerically increasing order. For example, if we run:

```
WeightedQuickUnionDS df = new WeightedQuickUnionDS(6); // creates WQUDS of size 6
df.connect(3, 2);
df.connect(1, 5);
df.connect(2, 5);
Iterator<Integer> connectedToThree = df.setIterator(3);
while (connectedToThree.hasNext()) {
    System.out.print(connectedToThree.next() + " ");
}
```

Then this code will print out: 1 2 3 5

Fill in the code below. Recall that a `DisjointSets` object stores items numbered 0 to $N-1$ (inclusive). For example if we instantiate new `WeightedQuickUnionDS(6)`, the valid item numbers for this object are 0, 1, 2, 3, 4, and 5. The `setIterator` method should take $O(N \log N)$ time. Note that you **may not create helper functions or new classes**. You may import any data structures you'd like (though this is not necessary) and you are not required to use the import that we provide.

```
import java.util.ArrayList;
import java.util.Iterator;
public class WeightedQuickUnionDS implements DisjointSets {
    ... // Instance variables not listed because you are not allowed to access them
    private int find(int p) { ... }
    public boolean isConnected(int p, int q) { ... }
    public void connect(int p, int q) { ... }
    public int size() { ... } // returns the number of items in the disjoint set
    /** returns an Iterator over the set containing p. */
    public Iterator<Integer> setIterator(int p) {

        List<Integer> res = new ArrayList<>();
        for (int i = 0; i < size(); i++) {
            if (isConnected(i, p)) {
                res.add(i);
            }
        }
        return res.iterator();
    }
    // YOU MAY NOT CREATE HELPER FUNCTIONS OR
    // NEW CLASSES.
}
```


7. Asymptotic Analysis (350 points).

Consider the code below. Assume that $h(N)$ runs in $\theta(N)$ time and returns a boolean.

```
static void alpha(int N) {
    if (N % 2 == 0) { // this check is done in constant time
        return;
    }
    for (int i = 1; i < N; i *= 2) {
        if (h(i)) {
            h(i);
        }
    }
}
```

a) alpha best (50 points). What is the best case runtime for $\alpha(N)$?

- $\theta(1)$
 $\theta(\log(\log N))$
 $\theta((\log N)^2)$
 $\theta(\log N)$
 $\theta(N)$
 $\theta(N \log N)$
 $\theta(N^2)$
 $\theta(N^2 \log N)$
 $\theta(N^3)$
 $\theta(N^3 \log N)$
 $\theta(N^4)$
 $\theta(N^4 \log N)$
 Worse than $\theta(N^4 \log N)$
 Never terminates (infinite loop)

b) alpha worst (50 points). What is the worst case runtime for $\alpha(N)$?

- $\theta(1)$
 $\theta(\log(\log N))$
 $\theta((\log N)^2)$
 $\theta(\log N)$
 $\theta(N)$
 $\theta(N \log N)$
 $\theta(N^2)$
 $\theta(N^2 \log N)$
 $\theta(N^3)$
 $\theta(N^3 \log N)$
 $\theta(N^4)$
 $\theta(N^4 \log N)$
 Worse than $\theta(N^4 \log N)$
 Never terminates (infinite loop)

Consider the code below. Assume that $h(N)$ runs in $\theta(N)$ time and returns a boolean.

```
static void beta(int N) {
    if (h(N)) {
        return;
    }
    for (int i = 1; i < N; i *= 2) {
        for (int j = N - i; j < N; j++) {
            int a = 1 + 1;
        }
    }
    if (N % 2 != 0) {
        return;
    }
}
```

c) beta best (50 points). What is the best case runtime for beta(N)?

- $\theta(1)$ $\theta(\log(\log N))$ $\theta((\log N)^2)$ $\theta(\log N)$ $\theta(N)$ $\theta(N \log N)$ $\theta(N^2)$
 $\theta(N^2 \log N)$ $\theta(N^3)$ $\theta(N^3 \log N)$ $\theta(N^4)$ $\theta(N^4 \log N)$ Worse than $\theta(N^4 \log N)$
 Never terminates (infinite loop)

d) beta worst (50 points). What is the worst case runtime for beta(N)?

- $\theta(1)$ $\theta(\log(\log N))$ $\theta((\log N)^2)$ $\theta(\log N)$ $\theta(N)$ $\theta(N \log N)$ $\theta(N^2)$
 $\theta(N^2 \log N)$ $\theta(N^3)$ $\theta(N^3 \log N)$ $\theta(N^4)$ $\theta(N^4 \log N)$ Worse than $\theta(N^4 \log N)$
 Never terminates (infinite loop)

Consider the code below. Assume $f(N)$ returns a random number between 1 and $N/2$, inclusive, and does so in constant time.

```
static void gamma(int N) {
    if (N <= 10) {
        return;
    }
    for (int i = f(N); i < N; i += f(N)) {
        gamma(i);
    }
}
```

e) gamma best (75 points). What is the best case runtime for gamma(N)?

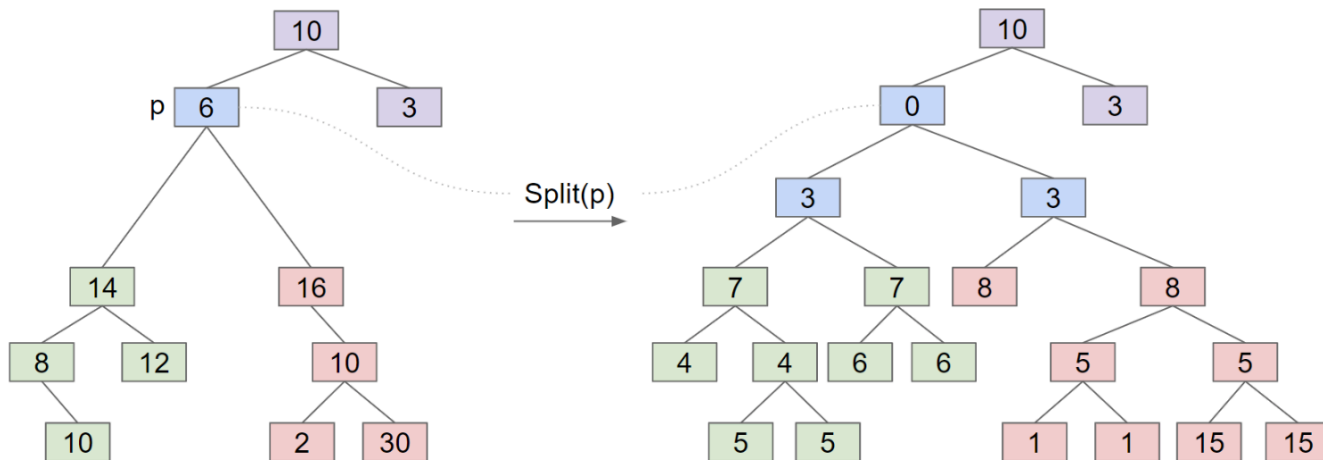
- $\theta(1)$ $\theta(\log(\log N))$ $\theta((\log N)^2)$ $\theta(\log N)$ $\theta(N)$ $\theta(N \log N)$ $\theta(N^2)$
 $\theta(N^2 \log N)$ $\theta(N^3)$ $\theta(N^3 \log N)$ $\theta(N^4)$ $\theta(N^4 \log N)$ Worse than $\theta(N^4 \log N)$
 Never terminates (infinite loop)

f) gamma worst (75 points). What is the worst case runtime for gamma(N)?

- $\theta(1)$ $\theta(\log(\log N))$ $\theta((\log N)^2)$ $\theta(\log N)$ $\theta(N)$ $\theta(N \log N)$ $\theta(N^2)$
 $\theta(N^2 \log N)$ $\theta(N^3)$ $\theta(N^3 \log N)$ $\theta(N^4)$ $\theta(N^4 \log N)$ Worse than $\theta(N^4 \log N)$
 Never terminates (infinite loop)

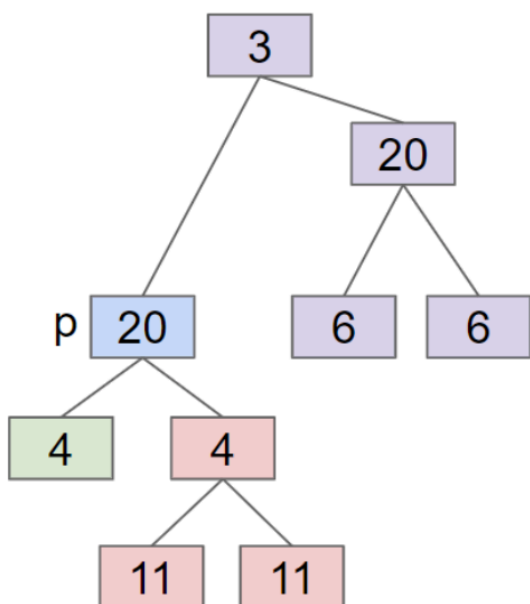
8. Recursive Trees (405 points).

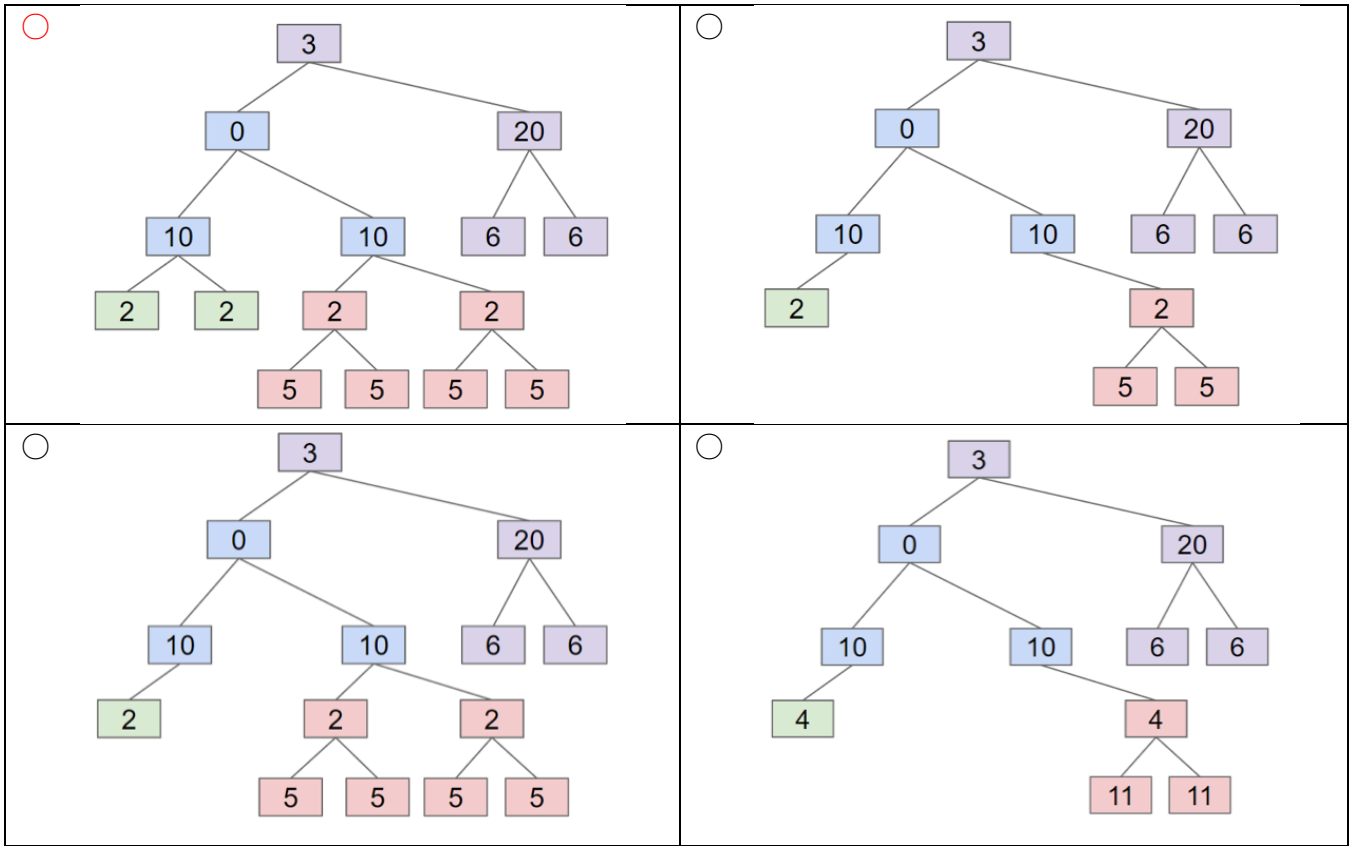
In this problem, you will write code that performs the split operation, as shown in the diagram below.



The split operation recursively splits every node X in the tree into two nodes of half the value, which we'll call XL and XR . XL keeps the left child of X , and XR keeps the right child of X . In place of the node which is actually split, a new node is created containing a zero.

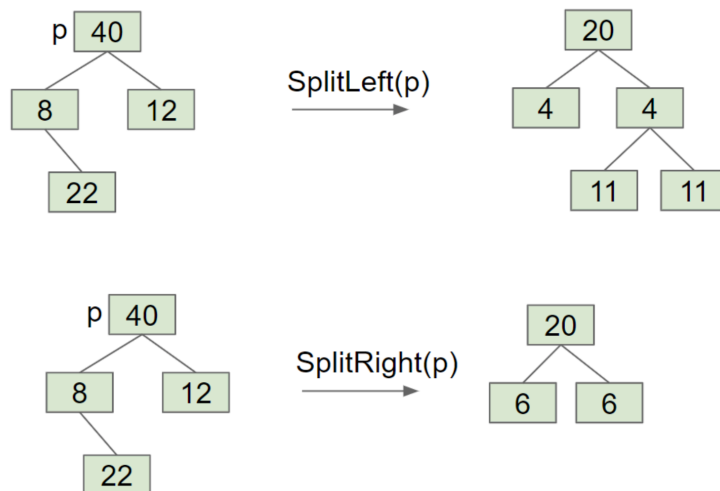
a) (65 points). As a warmup to parts b and c, if we called split on the node denoted as p in the figure below, what would be the resulting subtree? Assume that our tree stores integers, and that Java rounds down (for example $11 / 2 = 5$). Note: You'll get more out of this exercise if you try and do it first without looking at the multiple choice answers.





b) (220 points) As noted above "the split operation recursively splits every node X in the tree into two nodes of half the value, which we'll call X_L and X_R . X_L keeps the left child of X , and X_R keeps the right child of X ."

We call the tree rooted at X_L the "left split", and we call the tree rooted in X_R the "right split". Write the functions `splitLeft(Node p)` and `splitRight(Node p)` below. These should return the root of the left split and the root of the right split, respectively. As examples, consider the output of `splitLeft` and `splitRight` on the node denoted as p below.



The Node class definition is given below.

```
public static class Node {
    public int value;
    public Node left;
    public Node right;
    public Node(int v, Node l, Node r) {
        value = v;
        left = l;
        right = r;
    }
}
```

Your `splitLeft` and `splitRight` methods should not modify the tree passed to them. Instead, they should generate an entirely new tree. You cannot use the `split(Node p)` method in these methods.

```
public static Node splitLeft(Node p) {
    if (p == null) {
        return null;
    }

    Node newNode = new Node(p.value / 2, null, null);
    newNode.left = splitLeft(p.left);
    newNode.right = splitRight(p.left);
    return newNode;
}
```

```
public static Node splitRight(Node p) {
    if (p == null) {
        return null;
    }

    Node newNode = new Node(p.value / 2, null, null);
    newNode.left = splitLeft(p.right);
    newNode.right = splitRight(p.right);
    return newNode;
}
```

c) (120 points) Now write the `void split(Node p)` method. This method should modify the tree that it is given. For example, if we call `split(p)`, where `p` is the node containing 6 in the figure at the top of this page, after the call is complete, `p.value` should be 0, `p.left.value` and `p.right.value` should be 3, etc.

```
public static void split(Node p) {
    p.left = splitLeft(p);
    p.right = splitRight(p);
    p.value = 0;
}
```