CS 61BL Data Structures & Programming Methodology

Summer 2019

Midterm 2 Solution

This exam has 7 questions worth a total of 45 points and is to be completed in 110 minutes. The exam is closed book except for two double-sided, handwritten cheat sheets. No calculators or other electronic devices are permitted. Give your answers and show your work in the space provided.

Write the statement below in the blank provided and sign. You may do this before the exam begins.

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

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

Signature: Matthew Sit

Question	Points
1	5
2	7
3	12
4	0
5	6
6	6
7	9
Total	45

Name	Matthew Sit
Student ID	1234567890
GitHub account #	su19 - s1
Lab Section #	0 0 1
Name of person to left	Jackson Leisure
Name of person to right	Christine Zhou

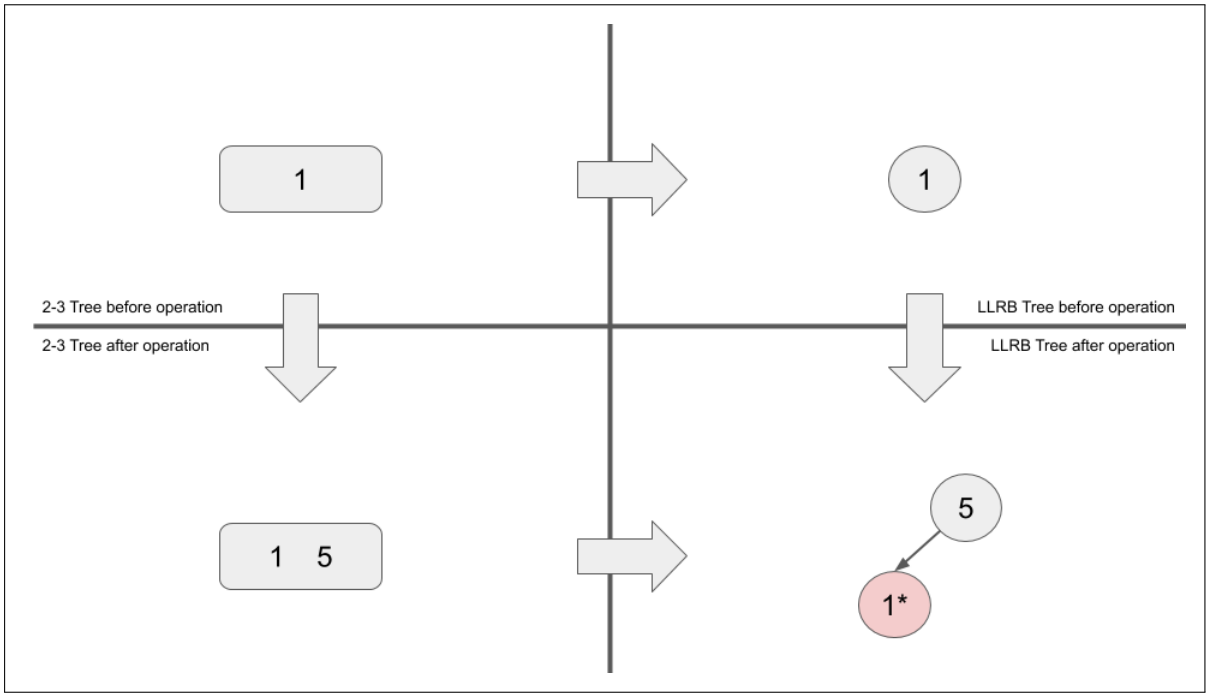
- There may be partial credit for incomplete answers. Write as much of the solution as you can, but we may deduct points if your answers are much more complicated than necessary.
- **Work through the problems with which you are comfortable first.** Do not get overly captivated by interesting design issues 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**. For code-writing questions, **write only one statement per line** and **do not write outside the lines.**
- Unless otherwise stated, all given code on this exam should compile. All code has been compiled and executed, but in the event that we do 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.'**

1. (5 pts) **Mechanical Balanced Search Trees**

   We learned that **2-3 Trees** have a one-to-one correspondence with **Left-Leaning Red-Black Trees**. For each subpart in this question, take the 2-3 Tree given in the left column and perform the indicated operation on it, drawing the resulting 2-3 Tree below the given tree. Then convert both the given 2-3 Tree and the 2-3 Tree you drew into LLRB Trees, drawing them in the right column next to their counterparts.

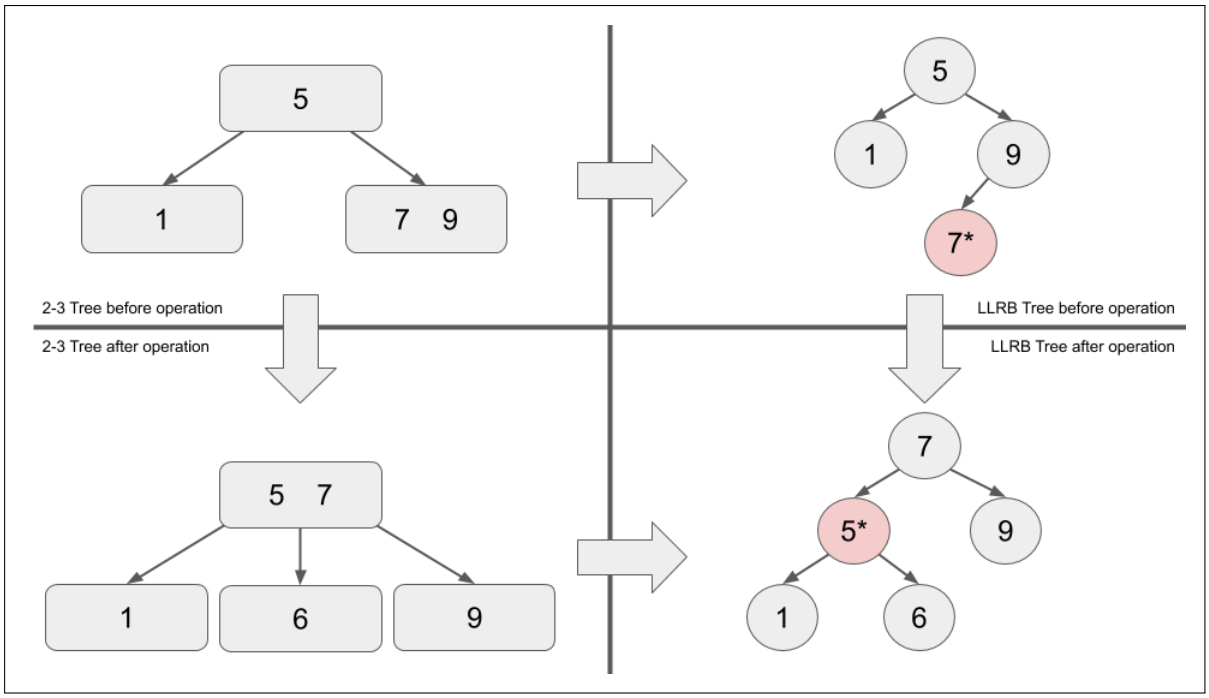
   Indicate a node is red by putting an asterisk (*) next to it. You must use the conventions and procedures we taught in lab this summer. Only your final answer within the boxes will be graded; perform any scratch work (which will not be graded) outside of the boxes.

   (a) `insert(5);`

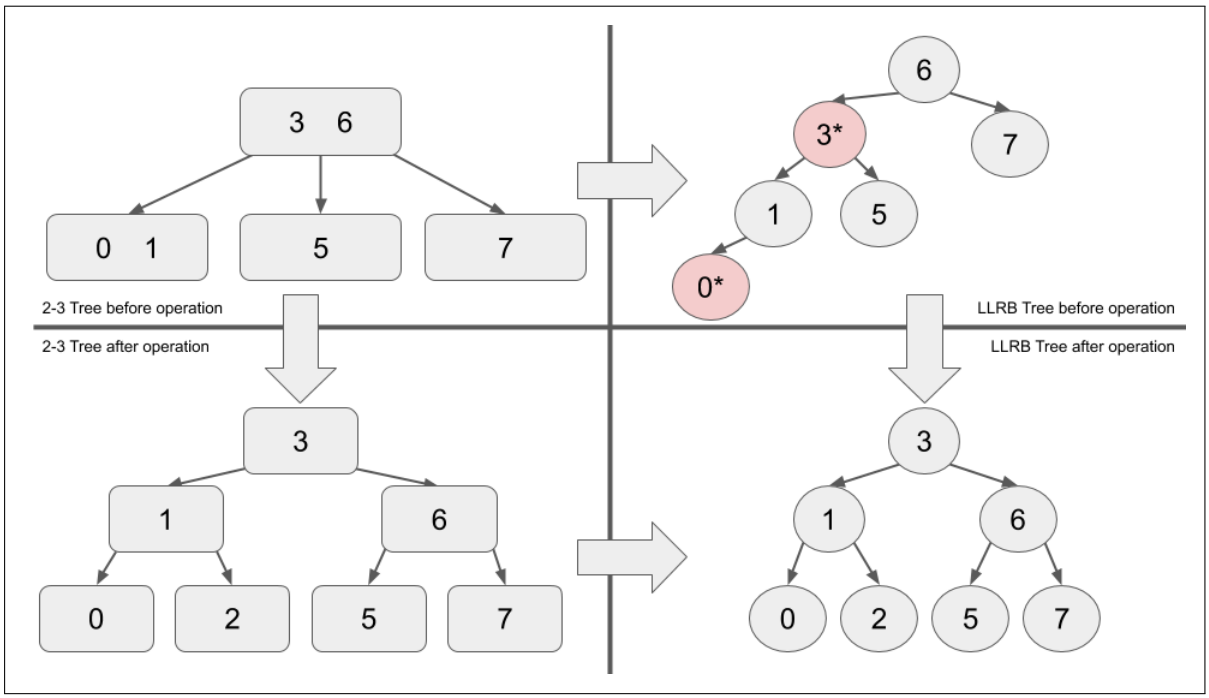


Name: Matthew Sit

(b) `insert(6);`

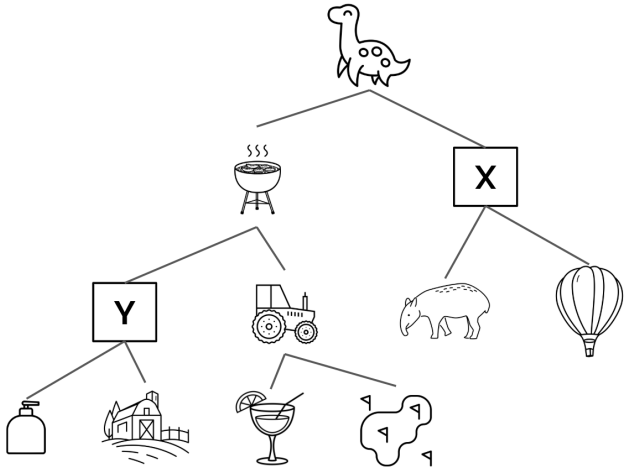


(c) `insert(2);`



2. (7 pts) **Heaps Spark Joy**

Shown below is a binary **Min Heap**, in which each symbol represents a node in the heap. The values contained within the nodes are comparable, and are **not necessarily unique** (except for the allowance of duplicate values here, all of the heap invariants from lab apply).



For each of the following questions, select all possible relationships that could occur between the values represented by the letter and symbol shown. **IMPORTANT:** For this entire question, assume that $X > Y$. Erase any undesired markings from boxes completely if you change your answer.

(a) Mark all that **could** be true of X:

☐ $X <$ 	☐ $X =$ 	■ $X >$ 
■ $X <$ 	■ $X =$ 	■ $X >$ 
■ $X <$ 	■ $X =$ 	■ $X >$ 

(b) Mark all that **could** be true of Y:

☐ $Y <$ 	■ $Y =$ 	■ $Y >$ 
■ $Y <$ 	■ $Y =$ 	■ $Y >$ 
■ $Y <$ 	☐ $Y =$ 	☐ $Y >$ 

(c) Representing the heap using the array representation from lab (in which elements begin starting at index 1), at what index will Y be located at? Write your integer answer in the blank below.

Answer: 4

Images courtesy of Google Noun Project.

3. (12 pts) **#BARTable**

Assume the standard `java.util.LinkedList` is imported everywhere as needed in this question.

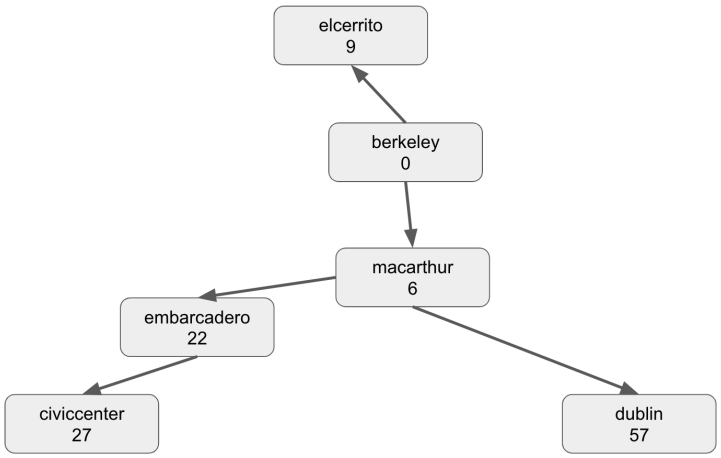
Bay Area Rapid Transit is the local rail and subway transportation system. From the nearby Downtown Berkeley station, you can travel to a number of exciting destinations in varying amounts of time. Our goal here is to print the names of all the stations, ordered by their travel times from Berkeley (from shortest to longest). A `Station` is represented as:

```
public class Station {
    String name;
    int travelTime;
    LinkedList<Station> children = new LinkedList<>();

    public Station(String name, int travelTime) {
        this.name = name;
        this.travelTime = travelTime;
    }
}
```

Note that the `travelTime` of a `Station` represents the travel time **from the Berkeley station** (not from another neighboring station).

We can build a tree of `Station`s rooted at the Berkeley station:



From this example tree, the expected result should be:

```
berkeley macarthur elcerrito embarcadero civiccenter dublin
```

Assume there are no cycles/loops in our rail system, and that every child station has a `travelTime` that is strictly greater than that of its parent.

(a) One simple way to solve this is to use a priority queue.

Implement the `print` method below using a priority queue. The argument `berkeley` represents the Berkeley `Station`, the root of the tree. Java's `PriorityQueue` constructor accepts a `Comparator`, which it uses to order its elements. **For this part, write a lambda expression to satisfy the `Comparator` argument.** Recall that a `Comparator`'s `compare` function "Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second."

```
public static void print(Station berkeley) {
    PriorityQueue<Station> pq = new PriorityQueue<>(

        (o1, o2) -> o1.travelTime - o2.travelTime                    );
    pq.add(berkeley);

    while (!pq.isEmpty()                ) {
        Station curr = pq.remove();
        System.out.print(curr.name + " ");

        pq.addAll(curr.children                        ); // pass in a List
    }
}
```

(b) The use of a lambda statement as the `Comparator` is a syntactical shortcut. Implement the following `StationComparator` which could be used instead.

```
import java.util.Comparator;
class StationComparator implements Comparator<Station>                    {
    @Override
    public int             compare(Station o1, Station o2                        ) {

        return o1.travelTime - o2.travelTime;                    
    }
}
```

Demonstrate how the `StationComparator` would be used in this line from the `print` function in place of the lambda that we used previously.

```
PriorityQueue<Station> pq = new PriorityQueue<>(

    new StationComparator()                                    );
```

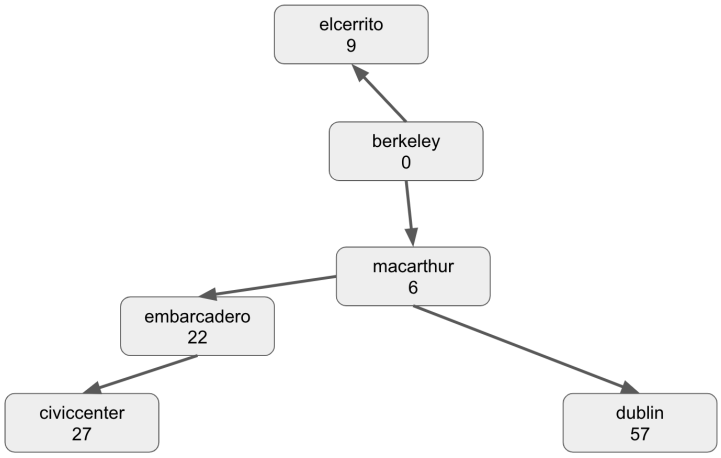
(c) What happens if the `@Override` line is deleted from above the `StationComparator`'s `compare` function, assuming that all of the blanks are filled in with the correct solution?

(Choose one answer.)

- ○ Compiler Error
- ○ Runtime Error
- ○ Unexpected Behavior
- ● No Difference

(d) Another way to solve this problem is to use a BFS-inspired approach. But before we fully implement that in part e, let's first remind ourselves of the standard BFS procedure.

Run BFS (Breadth First Search) as you learned in lab on the example tree from before (reproduced below), starting with the `berkeley` station. When inserting multiple children into the fringe, insert in order from lowest to highest travel time.



BFS Traversal (one station name per line):

berkeley

macarthur

elcerrito

embarcadero

dublin

civiccenter

In general, this doesn't always work in helping us print the station names in order of travel time, so your answer here may or may not match the example's expected output.

(e) A little trick resolves the problem we have with ordinary BFS: use dummy nodes such that the `travelTime` difference between any two consecutive nodes (whether a real `Station` or just a dummy node) is 1. For example, as shown below, since `sanbruno` has a child `sfo` that is 3 more away from it than its own travelTime of 122, we use two dummy nodes between them.

Implement `print2` to do the same thing as `print` but using this BFS approach instead. For your `fringe`, we will use a `LinkedList`, which can behave as either a stack or queue, depending on your usage of its `addFirst, addLast, removeFirst, removeLast` methods.

```
public static void print2(Station berkeley) {
    LinkedList<Station> fringe = new LinkedList<>();
    fringe.add(berkeley);

    while (!fringe.isEmpty()                              ) {

        Station curr = fringe.removeFirst();

        if (curr.name != null                             ) {
            System.out.print(curr.name + " ");
        }
        for (Station child : curr.children                ) {

            if (child.travelTime - curr.travelTime > 1    ) {

                Station dummy = new Station(null, curr.travelTime + 1 );

                dummy.children.add(child);

                fringe.addLast(dummy);
            } else {

                fringe.addLast(child);
            }
        }
    }
}
```

(f) Answer the following runtime questions in terms of the following variables:

- `N`, the number of official `Station`s (not including dummy nodes).
- `M`, the number of nodes in the tree, including both dummy nodes and official `Station` nodes.

Make sure to use the correct asymptotic bound notations (O, Ω, Θ) and to write them clearly (if you mean to write O, do not put a little loop at the top, because it may be interpreted as a Θ). Simplify your answers.

What is the runtime of the priority queue printing algorithm (`print` from part a), in terms of N?

$O(N \log N)$, $\Omega(N)$

What is the runtime of the BFS printing algorithm (`print2` from part e), in terms of M?

$\Theta(M)$

4. (0 pts) **PNH**

Name the type of bi-directional text in which alternating lines are read in opposite directions, similar to the direction an ox takes whilst plowing a field.

Boustrophedon Text

5. (6 pts) **Lost and Found**

A Lost and Found is a place where lost items are turned in and where people looking for their lost items can check to see if they've been found.

We are managing a Lost and Found, and we decided to use a `HashMap<Ticket, Object>` to store people's lost belongings. When someone finds a lost item and brings it to us, we create a `Ticket`, and store it and the item in the map. When someone asks us to look for an item they had lost, we make a new `Ticket` and use that to query our map. Since our lost and found just opened, the map is initially empty.

(a) To be able to use `Ticket` instances as keys in our map, we need to implement the `equals` method. Two `Tickets` should be considered the same if the `location` Strings spell out the same word, and if the provided function says the `description`s are similar enough.

```
public class Ticket {
    String location;
    String description;

    public boolean equals(Object o) {
        if (!(o instanceof Ticket)) {

            return false                    ;
        }
        Ticket t = (Ticket) o                                    ;
        // The next two blanks are for one long statement.

        return this.location.equals(t.location) && isSimilar(description, t.description);


    }

    // Returns true if the two argument strings are "similar enough".
    private static boolean isSimilar(String d1, String d2) {
        // Implementation not shown.
    }

    public int hashCode() {
        return location.hashCode() + description.hashCode();
    }
}
```

Name: Matthew Sit

(b) In one situation, a `Phone` was turned in to us. A `Ticket t1` was made for it and this pair was `put` into the map. When the owner came for it, a `Ticket t2` was made, and the `Phone` was successfully retrieved from the map using `t2`.

In another situation, a `Charger` was turned in to us. A `Ticket t3` was made for it and this pair was `put` into the map. After that, we found out that the `Charger` was actually found in a different location, and so we updated `t3` using a reference we still had to it (i.e. we used a line like `t3.location = ...`). When it was to be retrieved, a `Ticket t4` was made. `t4` should have matched the updated `t3`, but the map strangely returned `null` rather than the `Charger` as desired.

Why? Write no more than 25 words explaining this. Take note of the `hashcode` function from the `Ticket` class from part a.

The key was mutated and so was its hashcode.

The pairing is still assigned to the original bucket, and so cannot be found.

(c) Subsequent to the events from the previous part, one thousand `Shirt`s were turned in to us, and so we made a `Ticket` for each `Shirt`, and put them all into the map. After that, we again tried looking for `t3` using `t4`, and miraculously the `Charger` is successfully retrieved this time.

Why? Write no more than 25 words explaining this.

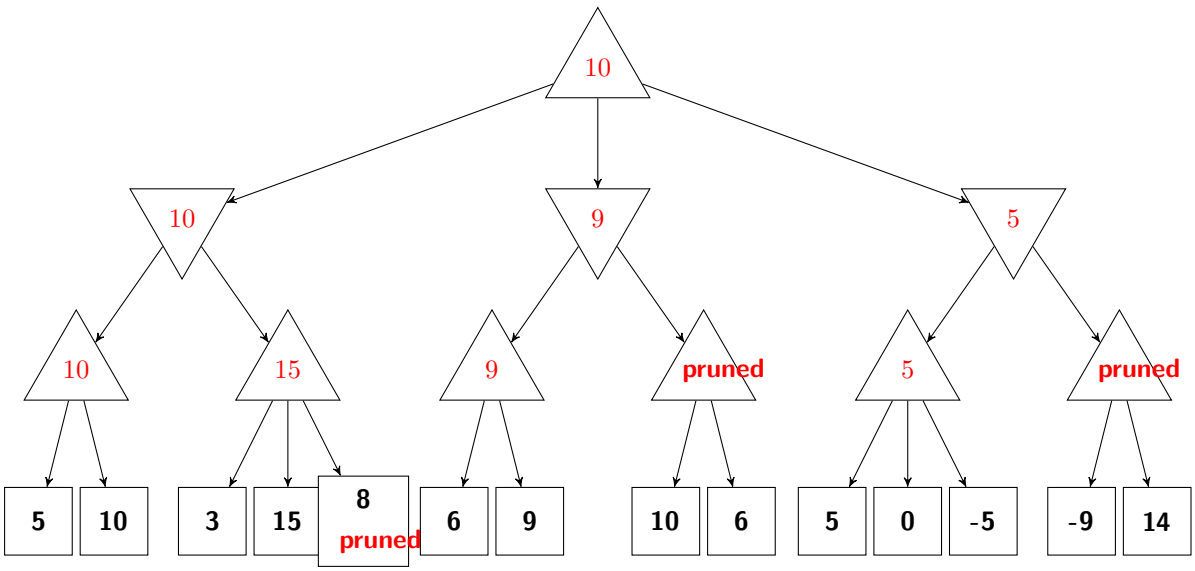
The addition of so many more entries probably forced a resize

of the map's underlying array, causing everything to be rehashed.

6. (6 pts) **Game Trees**

   (a) Given the following *mimimax tree*, **put an X on the branches** that would be pruned using *alpha-beta pruning*, and **fill in each node** with the value that it will return. The values in the nodes should account for alpha-beta pruning (i.e. do not write in a value that will never be seen).

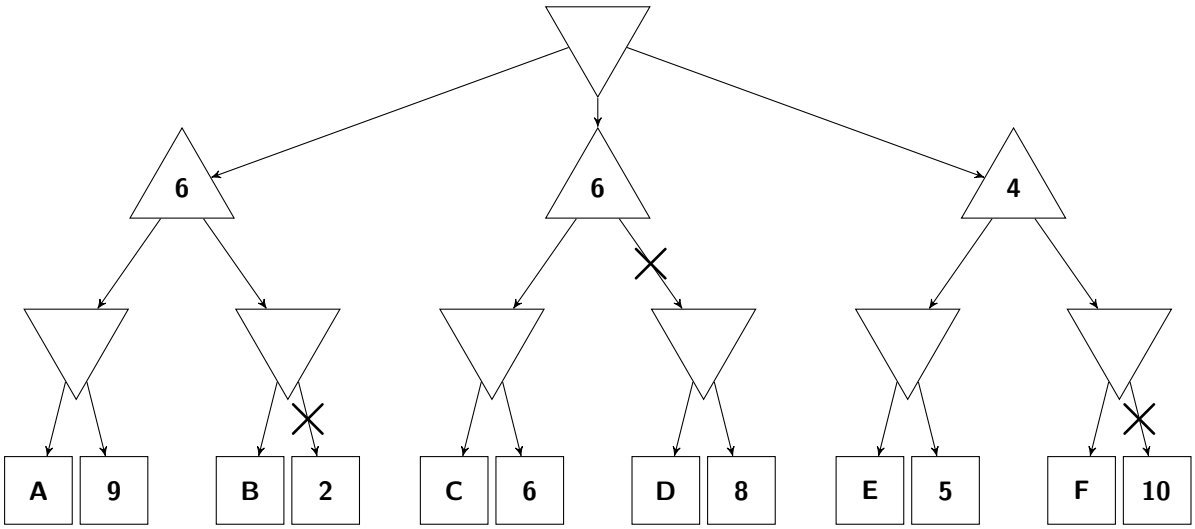
   The upward triangles represent maximizers and the downward triangles represent minimizers.



Name: Matthew Sit

(b) In the tree below, again the downward triangles represent minimizers and the upward triangles represent maximizers. The provided values shown on the non-leaf nodes are the values with alpha-beta pruning already performed. The branches with X's through them represent branches that have been pruned.

For each leaf node that is labeled with a letter, of the value options listed, mark all values for which it can be, given all of the provided values and also the branches that are pruned. You may make notes on this tree diagram, but it will not be graded.



(a) Mark the checkbox next to all of the values that Node **A** can be.
☐ 2 ☐ 4 ■ 6 ☐ 8 ☐ 10

(b) Mark the checkbox next to all of the values that Node **B** can be.
■ 2 ■ 4 ■ 6 ☐ 8 ☐ 10

(c) Mark the checkbox next to all of the values that Node **C** can be.
☐ 2 ☐ 4 ■ 6 ■ 8 ■ 10

(d) Mark the checkbox next to all of the values that Node **D** can be.
■ 2 ■ 4 ■ 6 ■ 8 ■ 10

(e) Mark the checkbox next to all of the values that Node **E** can be.
☐ 2 ■ 4 ☐ 6 ☐ 8 ☐ 10

(f) Mark the checkbox next to all of the values that Node **F** can be.
■ 2 ■ 4 ☐ 6 ☐ 8 ☐ 10

7. (9 pts) **Decision Tree**

Here you will implement a tree structure that is commonly used in machine learning since it naturally resembles an easy to interpret flow-chart. The non-leaf attributes and split values are hard-coded here, but typically these would be learned from training data.

Assume the standard `java.util.HashMap` is imported everywhere as needed in this question.

(a) Every day when lunchtime rolls around, Christine, Matt, and Jackson experience a different craving, which can be described by several `Attribute`s. In order for us to determine whether a restaurant will satisfy our overall craving, we need to be able to compare individual `Attribute`s of our craving and the `Attribute`s a restaurant can offer.

Fill in the blank below so that `Attribute` implements the `Comparable` interface.

```
public class Attribute implements Comparable<Attribute> {
    public String name;
    public int value;

    public Attribute(String name, int value) {
        this.name = name;
        this.value = value;
    }

    /**
     * Returns a negative integer, zero, or a positive integer as this Attribute
     * is less than, equal to, or greater than the specified Attribute in value.
     *
     * Assumes that the Attributes have names that are .equals().
     */
    @Override
    public int compareTo(Attribute o) {

        return this.value - o.value;
    }
}
```

(b) We will use `Attributes` in two ways. First we will put together a `HashMap<String, Attribute>` `craving` (passed in as the argument to the `predict` functions below, which you'll be implementing) that describes what we're looking for in a satisfactory meal. The `String` key of the map is the same as the `name` field of the corresponding `Attribute` values (for ease of use).

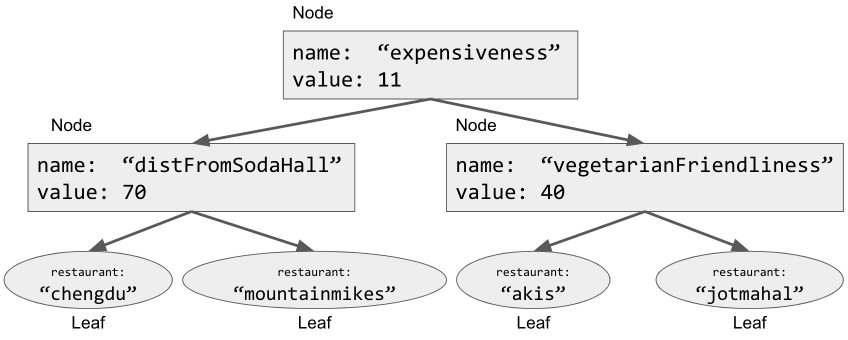
Secondly, we will be creating a binary tree out of non-leaf nodes (called `Node`, illustrated as rectangles below), and each of these store a single `Attribute` instance. The `name` of the `Node`'s `Attribute attr` will have the same `name` as one of the `Attributes` in `craving`, and we use its `value` as the node's **"split value"**.

We will compare the `Attributes` in the `craving` to those in the `Nodes` of the tree. During a call to its `predict` function, if said matching `Attribute` of the `craving` argument has a `value` that is **less than or equal to** that of `attr`'s split `value`, we look at the `left` subtree to recursively continue looking for our lunch destination. Otherwise, if the `value` is **greater than** the split `value`, we look at the `right` subtree. When a leaf-node is reached (`Leaf`, illustrated as ovals below), the `restaurant` field of that node indicates the prediction result to be returned.

Consider querying the example tree below with a `craving` consisting of the following:

- `"expensiveness"` to `Attribute("expensiveness", 12)`.
- `"distFromSodaHall"` to `Attribute("distFromSodaHall", 7)`.
- `"vegetarianFriendliness"` to `Attribute("vegetarianFriendliness", 40)`.

A call of `predict(craving)` on the root node should return `"akis"`. This is because the `craving` has an `"expensiveness"` value of `12`, which is greater than that of the `root` so we proceed to `root.right`. There, we see that the `craving` has a `"vegetarianFriendliness"` value of `40`, which is less than or equal to `40`, so we proceed `left` and return that `Leaf`'s `restaurant`.



continued on next page...

Fill in the blanks to complete the `predict` function implementations such that a provided `craving` will find its way to the appropriate `Leaf` and return the `restaurant` it best describes. You may not need all lines provided. No more than one statement per line.

```
public class Node {
    public Node left, right;
    public Attribute attr;

    public Node(String name, int value) {
        this.attr = new Attribute(name, value);
    }

    String predict(HashMap<String, Attribute> craving) {

        int c = craving.get(attr.name).compareTo(attr);

        if (c <= 0                          ) {

            return left.predict(craving);

        } else {

            return right.predict(craving);
        }
    }
}
public class Leaf extends Node {
    public String restaurant;

    public Leaf(String restaurant) {
        super(null, -1);
        this.restaurant = restaurant;
    }

    @Override

    String predict(HashMap<String, Attribute> craving) {

     return restaurant;

    }
}
```

(c) Answer the following runtime questions in terms of the following variables:

- `N`, the number of `Nodes` in the binary decision tree.

Make sure to use the correct asymptotic bound notations (O, Ω, Θ) and to write them clearly (if you mean to write O, do not put a little loop at the top, because it may be interpreted as a Θ). Simplify your answers.

What is the **best case** runtime of calling `predict` on the `root` of the decision tree and making recursive calls until we get a `String` answer? Note that the decision tree is not necessarily balanced. Assume that deletions are never performed on our tree (if this fact is relevant to your analysis).

$\Theta(1)$

The **worst case**?

$\Theta(N)$