

## Homework 2

Due: Wednesday, April 16th @ 11pm

As in Homework 1, a key part of this assignment is practicing debugging, this time in a more complex application. We'll ask you to submit a log describing all the time spent debugging.

Before you start, be sure to read the instructions for debugging to learn what information you need to keep track of while debugging. Then, as you work on each function, whenever you see a bug, open up your debugging log and record that information so that you can submit it.

If you get stuck during the debugging process, move on to the next section and come back later! Unlike Homework 1, you are not absolutely required to finish implementing the whole *app* before you debug, but you should try to implement a whole section before debugging it, to better distinguish implementation and debugging time.

**Note:** Please don't use `attu` (or another remote terminal machine) for these homework assignments as running a website and viewing it with `npm run start` is much better on `localhost`, where you can debug without the site being accessible on the broader internet, as it would be if you ran it on `attu`.

Check out the starter code for this assignment:

```
git clone https://gitlab.cs.washington.edu/cse331-25sp/materials/hw2-locations.git
```

Navigate to the `hw2-locations` directory and run `npm install --no-audit`. Then you can run the application with `npm run start` and open it at `http://localhost:8080`.



Name: Picnic

Color: red

The application displays a campus map along with 3 “markers” indicating things happening on campus. The App displays the current set of markers on top of the map. It also maintains a “selected” marker which is optional (as denoted with a ?), meaning it can have a value or be undefined. If a marker

is selected, this optional value will be defined and the `Editor` component is displayed below the map, allowing the user to edit the details of a particular component. Initially, this component simply displays the name of the selected item. You will write functions as described below to improve the functionality of the `Editor` to let users edit the name, color, and location of markers in various ways.

In order to quickly figure out what marker was clicked on (if any), the `App` component maintains the markers in a tree. This is similar to a binary search tree except that, because the lookup keys are  $(x, y)$  locations, nodes of the tree split into NW, NE, SE, SW quadrants instead of left and right. The details of the data structure are not important to the assignment (you won't use it in the code you write), but if you are curious, you can look in `marker_tree.ts` to see the details.

For your work, however, it is important to understand that the keys in this tree are  $(x, y)$  pairs, which have type `Location` (see `marker.ts`). As in a binary search tree, the keys are arranged carefully in the tree based on their relationship to each other. If a key is mutated, the tree may not work properly! (The same is true of `Maps` in Java.) In principle, `Locations` that are not being used as keys might be okay to mutate, but we strongly recommend that you take the safest approach and **avoid mutating** any `Location` objects in your code.

You'll notice that the `Editor` component includes the function `componentDidUpdate`. This is a built-in function of `React` that automatically runs (i.e. we don't need to manually call it) when the `props` or state of the component are changed. It takes the values of the props and state prior to the update which allows for comparing to find the exact change. In this case, we pass in props identifying the currently selected marker and initialize states of the `Editor` with those values, so when the props are updated for a new marker, we want to update the `Editor` states also. If the idea of props are unfamiliar, we encourage you to study up with lecture slides before you proceed!

For this assignment, we **ban the use of** `.map()` and `.forEach()`. In general, we recommend, for the sake of your debugging experience, that you stick to the functions and coding conventions we've seen in lecture.

## Locations App Implementation

### Updating Marker Name and Color

Update the Editor component to display the name of the marker in a `<input type="text">` element and the color in a `<select>` element. The list of available colors is provided in the `COLORS` array in `marker.ts`. Your UI should look something like this<sup>1</sup>:



A form for updating a marker. It has a text input labeled "Name:" with the value "Picnic". Below it is a select dropdown labeled "Color:" with "red" selected. At the bottom are two buttons: "Save" and "Cancel".

When the user clicks the “Save” button, you should invoke the `onSaveClick` callback passed in props. This will cause the App to re-render, showing the updated color on the map. When the user clicks the “Cancel” button, you should invoke the `onCancelClick` callback passed in props. This will remove the editor from the screen.

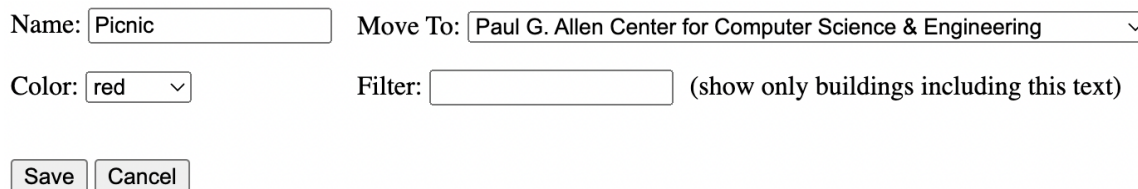
### Moving Markers

Update the Editor component so that the user can choose a building from a drop-down (an HTML `select` tag) and then click “Save” to move the marker to the location of that building.

Add an `input` text box that allows the user to filter the drop-down to only show buildings whose (long) names include the given text. Additionally, these filtered buildings should be ordered in the drop-down lexicographically (alphabetically). For example, typing “computer” into the box should reduce the dropdown to just the Gates and Allen buildings, with “Bill & Melinda Gates...” first and “Paul G. Allen...” second. When there is no text entered to filter on (in the initial state or if the user types a filter and then deletes it), the buildings in the drop-down should maintain their original ordering (as given in the `BUILDINGS` array).

Your dropdown should start with an option that, when selected, leaves the marker in the same location. (Without this, it would not be possible to change name or color without also moving the marker!)

Your UI should look something like this<sup>2</sup>:



A form for moving a marker. It has a text input labeled "Name:" with the value "Picnic". To its right is a "Move To:" label followed by a select dropdown showing "Paul G. Allen Center for Computer Science & Engineering". Below the "Name:" input is a "Color:" label with a dropdown showing "red". To the right of the color dropdown is a "Filter:" label followed by a text input and the text "(show only buildings including this text)". At the bottom are two buttons: "Save" and "Cancel".

---

<sup>1</sup>The precise details of the layout and styling are not important. This is not a UI design class.

<sup>2</sup>Again, the precise details of the layout and styling are not important. This is not a UI design class.

As before, the item should not move until the user clicks “Save”. “Cancel” should leave it unchanged.

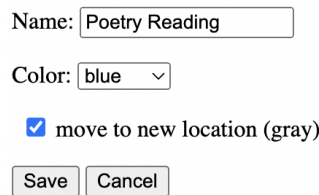
Make sure that, if you move a marker away from its original location, you can click on a different marker, click back on the first marker, and then move it a second time.

## Moving Markers by Location Click

Update the `Editor` component so that, when a marker is selected, the user can click on the map to move the marker to that location instead of selecting the name of a building.

The `App` component already detects when the user clicks on a marker and then clicks elsewhere on the app. In that case, it will pass in the new location in the optional `moveTo` property of the `Editor`. If a location hasn’t been clicked, the parameter will be `undefined`, otherwise, if that location value is provided, you should render a UI that allows the user to move the marker to that location or, by un-checking a box, not move at all.

Your UI should look something like this<sup>3</sup>:



Name:

Color:

☒ move to new location (gray)

In this case, the user should **not** see the UI you created in the “Moving Markers” section for moving to a building with a dropdown, and vice versa. In other words, the `Editor` should render the UI from this part if the `moveTo` property is defined, otherwise it should render the UI that you wrote in the “Moving Markers” section.

Again, the item should not move until the user clicks “Save”. “Cancel” should leave it unchanged.

---

<sup>3</sup>Don't make me say it again.

## Task 1 – A Log In The Machine

[100 pts]

In this task, you will try out your solutions to each section of the UI you added. As mentioned before, you can debug after implementing each section instead of waiting until the end, but don't forget to carefully **track** and **document** your time spent debugging.

For each bug, you must also provide the following information:

- What **failure**, (incorrect) app behavior, did you see that told you there was a bug?
- Which **experiments** did you perform to try to locate the defect? (Checking the network tab, scanning for typos, `console.logs`, etc.)
- What the **defect** was that caused that bug (if you ever found it)? (the line(s) of code)
- How many **minutes** did you spend on the bug after noticing the failure?
- Was the bug the result of *mutating* a field or array that was not supposed to be mutated?

Again, we have provided a [debugging log website](#) for you to record your debugging. Don't forget to save your log!

Once you have finished debugging your app, you will select *only* 3 log entries to turn in. Like Homework 1, you should try to select “interesting” entries, though you will not be penalized if some of your bugs were simple. However, **each log entry you select to turn in should have, at minimum, 1 experiment**. Experiments (your process) are the most important part of this assignment.

Your log should capture all the necessary context around each bug. Your TAs should be able to understand the UI interactions and inputs that led to the failure, and follow your experience through each debugging step. Experiments should start with some hypotheses with a leading question that you hope to answer with your experiment; then, upon seeing the result, we want to know what you learned which may lead to a next experiment (or to finding the defect).

It's understandable that you may hit some dead-ends and need to try a totally unrelated experiment idea, or that you may not find the defect, so it's okay if these show up in your log entries. Remember, that the goal here is that you improve your debugging skills, so try to make each experiment choice intentionally and avoid just “trying stuff” (but still log it if you do!).

You only need to turn in 3 log entries, but we encourage you to continue debugging your app to try to get all the behavior working because it's fun to have a working app! and it's good practice.

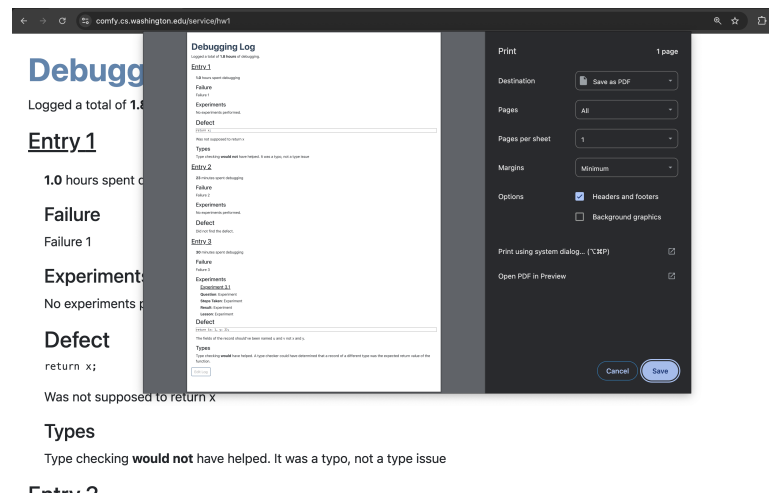
If you have been debugging for more than ~6 hours and have yet to find 3 bugs to turn in, we *highly* encourage you to reach out to the staff for help! Come to office hours or make a private ed post and let us know what's going on, so we can try to give you some extra debugging support. Sometimes bugs takes days or weeks to debug, in the “real world”, but *extremely* time-consuming bugs are not the intention for this class, so make sure you're getting our help if you need it!

**If you think your implementation is correct, and you have not found 3 bugs to log**, or you have not found 3 bugs that required experiments (e.g. immediately obvious typos), you can send us your implementation, and we will return it to you as soon as possible with new bugs for you to debug. To send us your work, you should upload `Editor.tsx` containing your completed component to Google Drive, configure the share settings so we can access it by link, and email that link to [cse331-staff@cs.washington.edu](mailto:cse331-staff@cs.washington.edu). If you need to send us your implementation for bugs, you **MUST** do so by **Monday, Apr 14th at 7pm**.

## Submission

After you finish debugging:

1. Open each log entry that you want to include in your submission and select Show in “View” box.  
**Show:** ☒ (in “View”)
2. From the main page, select “View Log” to open all of your selected log entries.
3. Select File > Print or ctrl+P/Command+P to open the print dialog.
4. Set the print destination as “Save as PDF” and “Save”



5. Submit your downloaded log and modified Editor.tsx to the “HW2” assignment on Gradescope.

**Note:** Remember that your Debugging Log should have 3 interesting bugs logged. Also, the code you are submitting in Editor.tsx does **not** need to be fully functional or correct to receive full credit, but there should be a meaningful attempt for each function. This is because the focus of this assignment is debugging!