Summary

Challenges

We found the most challenging parts to be insert and remove withing the 2-3 tree implementation. This was because the way we wrote it, we had to understand how the 2-3 tree worked, and then had to exhaustively pattern match against all cases, while simultaneously try to group cases together in order to reduce the amount of code to write.

Design Decisions

One interesting design decision that we made was to implement insert and remove separately. While we could have written insert by first calling remove to make sure the item we were inserting in the tree was not there, and then writing an insert function that inserted items that we knew wasn't in the tree, we decided to make them completely separate. This was done in order, to first make sure that if remove had any errors in it, the insert function would still work, and second, to make insert more efficient, as if the key was still in the tree, we could just replace the value.

Major Issues

None

Known Bugs

```
 let dictv1 =
empty |>
insert 3 4 |>
insert 6 4 |>
insert 29 4 |>
insert 15 4 |>
insert 2 4 |>
insert 4 4
let dictv2 = remove 3 dictv1
```

returns a fatal Unreachable exception. I was not able to figure out what was wrong with this, but I suspect the problem is either that insert is not inserting things correctly, or more likely remove is not

Design and Implementation

While most of the design of the project was given to us, in order to implement it, we made some key choices. Firstly, whenever we could, with the exception of not using delete in insert for the 2-3 tree implementation, we made sure that if a part of of one function could be used to implement another, we did so in order to reduce the amount of duplicated code. We did a top-down approach, where we first defined what the data structure was, and then made sure that, given the tree was given in the correct structure, each function did what it was supposed to do. This comes with numerous advantages and disadvantages. Specifically, each function is very short and clear – it takes in a certain input and returns the correct output. However, it is very sensitive to bad input because of this – besides the functions rep ok to test whether an input is correct, each function does not test the structure of inputs, and therefore if given bad input, there are no guarantees of what happens. For example, in data.ml, I implemented an exception called Unreachable. This was done so that all pattern matching was complete, but given good input, the function would never reach that part of code. Thus, this gave a good cue that if ever the exception Unreachable came up in testing, I either gave the function bad input, or the function was wrong.

Testing

We decided to not using huge files in the interest of time - but still tested on them to ensure that our implementations worked. We worked to ensure that testing was complete – especially on functions that were

more complicated, like insert or remove, while is empty and empty did not need quite so thorough testing. We tested exceptions in utop to make sure they were thrown. We did attempt to use assert raise, but in the interest of time, did not add all of these cases to our test files.

Work Plan

The way we divided up the work, was for George to work on all the functions in data.ml, the basic list implementation, the set dictionary, and then the dictionary implementation using 2-3 trees, while Eric worked on making sure engine.ml was functional, as well as doing a majority of the testing. This minimized the risk of redesign, as each person had full control of the modules that they implemented, thus design decisions would not clash as any sort of specifications that each person decided on for each part would not affect the other. Some key dependencies of doing it this way was that Eric could not test parts of engine until the set dictionary was implemented, and this could have created a problem where bugs and issues with that would not have been revealed until later.

Comments

We spent approximately 20 hours total working on this project. Within data.ml, approximately half was dedicated to designing (specifically designing our implementation of insert and remove from a 2-3 tree so that it correctly models the behavior of a 2-3 tree) and then half was dedicated to coding and testing now that the behavior of these functions were clearly understood. Some advice that you should have given before we started was how to access and deal with the modules. Once the functions were written, we had trouble instantiating some of them. What was hard about this assignment was working with a partner, making sure that we split up the work in such a way that using git was never a problem – making sure that we coordinated such that each file was only being edited by one person at once, and ensuring that we were not waiting on each other for certain functions to be done. In addition, if one person had a bug in their implementation, and it was giving the other person errors, this was hard to resolve. What was surprising about this assignment is that it was actually very easy to divide the work – as long as each person knew what had to be done, because of how segmented it was, we successfully got it done.