# Debugging

Although debugging is not strictly a topic in CSE 332, the programs you will be writing are large enough that efficient debugging will be essential. We realize that many of you have never debugged programs as large (or complicated) as the data structures you will be building, so we're hoping this handout will lead you down the right path.

## Debugging "Non-Strategies"

This section describes strategies that you should *absolutely avoid* when debugging. They lead to extra frustration, often don't help you find the bug, and won't work at all as the programs get larger and more complicated.

### Stare and Hope

When you have a complex program... spread out over multiple files... with some code that you didn't write, it's impossible to keep it all in your head. A common attempt at debugging is to *stare at the code* and wait for the bug to "pop" out at you. In CSE 143 level programs, this often works, but it's not a good approach from now on. The intuition on why this doesn't work well is that you're not looking at the states the program gets in–computers are better at being computers than we are. So, you should always be running your program to debug.

### "Shotgun" Debugging

A "slightly more advanced" strategy is to make a change that "feels right", then run the program to see if it worked. Then, make a change, then run the program, etc. Although this time you're running the program, The major concern with this strategy is that every time you make a change, you're not *learning anything about where or why the program went wrong*. Every time you run the program, you should be ruling out some potential reason the program is broken.

## Debugging Tools

While the approach you take to debugging is the most important thing, the tool you use can also make-or-break your debugging session. We recommend you try both and use whichever one makes more sense in the situation.

### `println`

One way of examining state is to... print out the state. One "gotcha" with our testing framework is that you have to use `System.err.println` (rather than `System.out.println`). Whenever you print out state while debugging, you should "mark" it with some string; this way, you know where/when in the program the output is from. A really common way of debugging with `println`s is to surround the potential problem area with a "begin" and an "end" and attempt to narrow down the area that the bug occurs in (more on this later in the handout).

### IDE Debugger

Another way of examining state is to use the built-in debugger in whatever IDE you have. If you're going to do this, please make sure to read the tutorial before using it. Make sure you understand the difference between "stepping over" and "stepping in". The debugger can be used in much the same way as the `println` statements, but the approach is different. Instead of surrounding the area that might be broken, you step through until you hit the place it is broken. Unfortunately, you will often have to step through the program multiple times even after you've found the error. The reason for this is that you'll need to step more carefully to figure out *exactly* where things went wrong.

## Debugging Strategies

There are a wide variety of approaches to debugging, and you should learn any and all of them. You'll likely end up having a "favorite" approach, but we recommend varying the approach based on the type of bug you've run into.

### Tell A Story

A bug is nothing more than a divergence between your expectations (the "story" of what your program is supposed to do) and reality. Debugging is reconciling *the exact place that the stories diverge*. If you can find the place where they diverge, then you can understand what went wrong by examining state right before and after the divergence.

To find the moment of divergence, you should attempt to narrow down the possible scope of the bug. To put it another way, before you start debugging, the error could be anywhere from the first line of `main` to the last line of the program. You can usually very quickly narrow down the scope to a much more reasonable piece of the program. The hard part is narrowing it enough to see exactly where the bug is.

For this, we recommend doing (effectively) a binary search. That is, see if "around the middle of the program" shows the buggy behavior: if it does, then search later; if not, then you can move the end of your scope to the middle. Then, you keep on narrowing until you've found exactly where the bug is.

### Run An "Experiment"

Debugging can be treated like a scientific experiment. That is, before running the program, you should make a hypothesis as to what the bug is. Then, design a way to confirm or deny your hypothesis. Finally, run your program and actually confirm or deny your idea. If you weren't right, then come up with a new hypothesis. The important idea here is that you should never edit your program without some sort of idea of where or what the actual bug is.

### Fail Fast

Some of the most difficult bugs happen because there is an error early in the program that doesn't show up until significantly later in the program. For example, if you `insert` lots of things into a data structure, you don't actually notice any errors until you do a `find`. Debugging is significantly easier if you make your programs "fail fast". All this entails is checking that all the invariants are met at the beginning and the end of every method. In other words, you should write a `private` method that "checks" the internal structure and if it ever returns false, throw an exception and end your program.

### Take A Break

If you've been debugging for a while and you're stuck, *stop debugging*. Take a break. Do something else. Anything else. The more frustrated you are, the less productive and efficient you will be.

### Ask For Help

If you have a particularly thorny bug, you've tried all of the above, and you have no idea what's going on, *come talk to us*! We will help! We can walk you through debugging strategies. (We won't do it for you, but we'll do it with you if you've made an attempt.)

# Common CSE 332 Debugging Issues

## General

The following items are issues you might run into during any (or all!) of the projects in CSE 332.

### The Eclipse Debugger doesn't work with the provided tests

This happens because the testing framework does some fancy things to make sure different tests don't interfere with each other. If you want to use the debugger, copy the actual content of the tests to your own testing file and use that `main` instead.

### I keep on getting a `ClassCastException`

If the error is something like

```
java.lang.ClassCastException: [Ljava.lang.Object; cannot be cast to [Ljava.lang.Comparable,
```

this means that when you are creating an array (probably the one backing your data structure), you are using the wrong type. Take a look at the generics handout, and make sure that you're following it exactly.

### I'm updating the size/root/etc., but it's not changing

It's likely that you're "shadowing" a variable. The super classes you're given have fields for `size`, `root`, etc. If you create your own in the subclasses, then your data structure has two conflicting roots. Make sure to use the one in the super class instead.

### The tests are failing, but I'm pretty sure my data structures are working!

Make sure you're updating the `size` for your data structure. Also, make sure you're throwing the right exceptions based on what the super class says to do.

### The tests time out on my computer, but not on gitlab-ci

The machine that we will be testing your code on is similar to the gitlab runners. Your personal machine may be significantly less powerful, and that's okay.

### The tests fail on gitlab-ci due to a `ClassNotFoundException`, but not on my computer

Scroll to the top of the test output on gitlab; it should indicate what the issue is and how to fix it.
(The Eclipse compiler is more lenient about generics than `javac`, which is what the gitlab runners use.)

## P2

**testRepeatedWordsPerNGram is failing, but 3 == 3**

You're likely creating a "new Integer", but you should be using the one you were given.

**Cannot instantiate the type Dictionary<AlphabeticString, Integer>**

Dictionary is an abstract class. So, you can't make a new one (which type of dictionary would it be?). You should use newInner and newOuter as if they were constructors for dictionaries. If you carefully read the spec, it explains exactly how to do this.

**NullPointerException in MoveToFrontList**

Make sure you're using .equals instead of ==

**NullPointerException in NGramToNextChoicesMap**

Make sure that your `HashTrieMap` is updating its `size` field correctly when changing the value of items that already exist in the `HashTrieMap`.

**The Iterator for BST (or AVLTree) seems broken**

Your worklists should be capable of storing nulls.

**The type E is not a valid substitute for the bounded parameter <E extends Comparable<E> > of the type MinFourHeap<E>**

You forgot to remove the "extends Comparable<E>" from the top of your MinFourHeap.

**My ChainingHashTable puts identical items in the dictionary multiple times**

Remember that hash tables rely on the equals method; you should make sure you're comparing the keys, not the items. Also, make sure you're using .equals and not ==

**HeapSort and TopKSort pass the tests locally but not on gitlab**

Make sure your MinFourHeap is initializing arrays of Object rather than Comparable.

**My ChainingHashTable passes the gitlab tests, but when I run uMessage on a large corpus, it doesn't work**

Make sure that you're handling the case where the table gets really big–the corpus might be larger than the sizes you've provided.

**uMessage gives a NullPointerException when I type really fast and it tries to autocorrect**

Don't worry about this; it's not your fault.

## P3

**Minimax times out on gitlab-ci!**

Make sure that you're not computing things you don't need to. For example, if you hit the base case: do you need to compute all the moves? Do you need to copy the board?

**AlphaBeta times out on gitlab-ci!**

See the minimax advice above. Also, make sure you're actually passing an **updated** value of `alpha` to the recursive calls; if your code passes `alpha` to the recursive calls but only ever updates `best.value` (or some other variable), you are effectively never pruning anything.

**AlphaBeta doesn't work when I use `BestMove` objects to store `alpha` and `beta`, but does if I use `int`**

`BestMove.negate()` is a mutator; it negates the value of the `BestMove<M>` object it is called on, and returns `this`. So if you pass `alpha.negate()` to your recursive call, you are *also* negating the value of `alpha` in the current call as a side effect.

**My bot doesn't work when I initialize `alpha` as `Integer.MIN_VALUE`**

Consider what happens when you compute `-Integer.MIN_VALUE` as part of your recursive call. You should use `-evaluator.infty()` instead.

**When I run my bots, I get a `ConcurrentModificationException`**

The order that you copy a board, make a move on that board, and recurse is very important. You are likely modifying the same board in two threads. This can happen if you're not copying it in one of the threads (e.g., the one you "compute" on). It can also happen if you make a move before copying.

**My bot is returning moves that don't appear in `board.generateMoves()`**

Make sure that you always return a move from `board.generateMoves()` (which contains all valid moves for the current player), bearing in mind that `board.applyMove()` changes which player's turn it is.

**My bot appears to be running forever (even after increasing the timeout)**

Make sure your code handles the case where `generateMoves()` yields a list of size 0 or 1, is not trying to `join()` any `compute()`'d tasks, and is not passing arguments in the wrong order (e.g. passing `hi` as `depth`).

**My Jamboree only works with `PERCENTAGE_SEQUENTIAL` $\in$ {0, 1}**

Remember that the best move can be in either the sequential *or* parallel section of the moves list; your code likely merges these partial results incorrectly. In particular, you should break ties in favor of the sequential result.

**My Jamboree appears to only ever do 0% or 100% of the moves sequentially, and is very slow**

If you have the following line in your code:

```
split = (int) PERCENTAGE_SEQUENTIAL * moves.size();
```

Java will parse this as:

```
split = ((int) PERCENTAGE_SEQUENTIAL) * moves.size();
```

because casts bind stronger than multiplication. This makes your code slow and hides merging errors (see above).

**My bot passes the tests, but it can't beat `calculon`!**

Make sure you've increased the search depth in `Engine.java`. Also make sure that your base cases (particularly `moves.isEmpty()`) are correct, otherwise your bot may deliberately avoid winning.

**`TestGame` doesn't stop at a stalemate!**

Yes. This is expected behavior. If you reach a stalemate, you can stop the program. `TestGame` is meant for testing–not for actual playing.