# ASSIGNMENT 3: SEARCH

KYRYLO CHERNYSHOV (KC875), ROQUE SOTO (RAS656)

## 1. Summary

Testing was rather challenging. Aside from knowing that our test cases will themselves be tested through the harness, another challenge was dealing with the complicated project structure and encapsulation. Eventually, we did get it working. An interesting decision that we made was to make the int in the AVL type carry the height of the tree, and not the balance factor, which is what we wanted to do earlier, and perhaps was the way the course staff wanted us to use it.

## 2. Design, implementation and testing

2.1. **Design decisions.** An important design decision that we made was, as mentioned, was our use of the int in the AVL type. Our reasoning was that computing the balance factor given two subtrees now became trivial, and we could update the height every time we did an insert, remove, or a tree rebalance.

Another decision was to implement AVL trees using the handout's rather abstract way, instead of the way we wanted to previously, which would bubble a rotation function up each node after an insert/remove. The two ways are equivalent, but the abstraction made the former much simpler to implement.

2.2. **Helper functions.** We implemented some very useful helper functions. One of them was height, which at first recursively calculated the height of the tree, but - later on, after our change to the interpretation of the AVL type - simply read off the value of the height if the tree was a Node, or returned 0 if it was a Leaf. This function proved to be very useful, and saved us quite a few lines of code. We also implemented le and ge, functions that compared two values, mapping the anonymous variants EQ, LT and GT to booleans. For rep_ok in MakeTreeList, we implemented is_bst and is_avl, which checks the BST and AVL invariants, respectively. In Engine, we implemented a few useful helpers, such as list_of_matched_strings, which takes a string and a regex, and returns the list of all substrings that are matched by the regex; this was useful since OCaml does not provide this natively. Finally, perhaps the most important helper functions are ins_fix and del_fix. These two functions are bubbled up the tree in the case of an insert or remove operation, respectively, and rebalance each subtree if needed. The reason we needed different functions for this is that the possible rebalancing cases were slightly different with insert and remove. We implemented these functions with a painstakingly annoying pattern-match.

2.3. **Testing.** We are fairly confident in our testing, purely because we have so many test cases in the harness. I think making us do the harness did wonders for our confidence in our code. Most of the time, when we ran our test cases they all passed. This is because pretty much all bugs were discovered by manual testing in utop: every time we implemented a function, we would test it for a bit in utop, and only then write physical test cases. We also had a few test cases that used our own local files; these could not be submitted as per the rules. Also, in some test cases we used functions that we had already tested above to test other functions.

2.4. **Efficiency.** Somehow, our ListEngine can index the test2 directory in about 10 seconds. This is very different to the stated approximate time of 2 minutes, but we checked this with our TA and apparently it has been done before, and we have tested MakeListDictionary extensively. We think this is because we wrote very efficient list and set processing functions. For example, we used List.sort_unique to filter out unique elements, and we think this was a good decision because 1) less code for us to write :) and 2) OCaml's native sorting is probably highly optimised, much better than what we could have done. Sorting sounds like an unnecessary step in filtering unique elements from a list, but it is actually the most efficient way: sort, and then traverse the list to remove duplicates. List.sort_unique does both steps, in $O(n \log n)$.

Another function that we did well was set intersection. We considered doing a naive $O(n^2)$ implementation, but decided on taking more time and writing it in $O(n \log n)$.

## 3. Division of labour

I (Kyrylo) physically wrote this overview, but my partner Roque had significant verbal input. As for the code, we decided upon a divide-and-conquer strategy. Some code we would implement together, typing on the same PC, and sometimes we would implement two closely related functions by taking one each. For example, for tree insertion, I implemented insert but we implemented ins_fix together, but for tree removal, Roque implemented del_fix, while I implemented remove, which calls del_fix. Roque took on the gargantuan task of handling the test suite and harness, while I did most of the code in Engine. At the very beginning, we implemented most of MakeListDictionary and MakeSetOfDictionary together, but we did some of the functions separately. For example, we shared union, I took intersection and Roque took difference. Our teamwork was so good that we even shared bugs, with each of us producing about the same number.

## 4. Known problems

We're not sure how much of a problem this is, but when we compile our code, there are a few warnings regarding "illegal backslashes" in the regex that we use to match words in Engine. As far as we know, backslashes are needed for regex constructs like groups () and or —. The code works, but we get the warnings anyway, likely because OCaml doesn't recognise the string as a regex, since it is not a built-in type.

## 5. Comments

We enjoyed this assignment, although it felt a bit counterintuitive implementing a set from a dictionary. We think it would have made more sense to implement MakeListSet and MakeTreeSet, and then MakeDictionaryOfSet, which would use key-value pair tuples as set elements.

We spent quite a lot of time on this assignment: just over a week of work. Some good advice would have been to really make sure we start early for this assignment. We were quite lucky to have finished in time.

Obviously, we found trees to be the hardest part, but it was also quite fun once we figured out how to do it. The handout helped an immense amount in that regard, however we noticed that the avl handout had a typo which caused a bug and made us unsure of what was wrong with the program. Specifically, the specification for delFix's L2 case on page 6 of the handout which suggests that tree a's height changes even though the tree remains unchanged.