Design doc.

Alexader Yermakov [OUI 20407 DSA Course]{26/06/2019}

MajellaMap

- Open addressing HashMap
- Can use single values as input to behave simply like HashSet *(i.e. hash table, as was requested)*
- Uses MurMur3 hashing algorithm
- Uses Robin Hood hashing *(trying to fit items as close as possible to their best position)*
- Uses Fibbonachi mapping of hash values *(to get more even distribution and reduce preliminary expansions)*
- Limits the amount of maximal scans to log2n of current size, if reached expands
- *Currently cannot shrink if already expanded*

This project can...

- Use text and dictionary as input *(via command line args)*
- Load text to internal RedBlack Tree data structure
- Load dictionary to HashTable styled HashMap via MajellaMap
- Run though Mapped dictionary and remove all the occurenses of its word in the Tree
- Print the result in readable format *(examples in sample_output.pdf)*

Known limitations

- Although has a default dictionary and demo file structure, I couldn’t make it run when packaged to Jar. :(
- That means it HAS to get two arguments as input [text_file] and [dictionary_file]
- Uses regex to remove puctuation from splitted singular words that mean something like "*(apple*" will get transformed to *apple* and pass the check, technically in the scope of the assignment it doesn’t matter. *(it was an attempt to generify the pattern matching, so I can split it to multiple stages in the future)*.

INSERT/DELETE Complexities

On all **put()** operations, we get **lo2n** worst case scenario.

So we get **lo2n** amortized complexity, if we don’t count the map reallocation when it expands.

1. By using Knuth’s multiplicative method *(Fibbonachi mapping)* we get more or less even distribution of hash values, when they get reduced to lower range *(current map size)*.
2. On each addition we caculate item position and see if someone already occupies this place.
3. If yes, we compare its *povertyLevel* to our, that is basically a characteristic that represents how far away from the best position this item is currently sitting.
4. If item is wealthier than the one we trying to insert, we steal its

position and continue inserting the item we stole the place from.
5. At most we will do it **lo2n** cycles of this loop, at which point we will force expansion and rehashing.

On all **remove()** operations, we get **lo2n** average case scenario.

1. The to actually find the item we want ot remove is still **lo2n**.
2. However after finding we will proceed to *fix* the Map, from the point we removed the item, to it's end, up until we hit some item that is already slotted in its best place, or an empty cell.
3. Current MAX_LOAD_FACTOR is 0.53, so it is practially impossible for us to get in to situation, when we need to readjust **n** items form the first cell to the last, for that we would have to have MAX_LOAD_FACTOR of 1.
4. Since we readjust the items and try to distribute them evenly even with bad hashing function we will expand before any such such scenario can happen.

Implementation details

- Our MajellaMap size is growing in powers of two + log2 of its size, however from mapping function point of view its size is just two in some power, without adding its power to final result.
- For that reason it will never spill out of bound.
- We store the temporarily stolen items if last slot of the Map, it can never be mapped, for reasons above.
- Checking if this slot is empty or not by the end of insertion cycle, tells us if we need to expand or not, however LOAD_FACTOR can

affect this desicion too.

- I used 12Dict collection of dictionaries to test the spell checking, they are included with rest of the files.
- As a secondary text for tests, I’ve used first three chapters of The History of the Peloponnesian War, its also included.

Used tech.

- IntelliJ Community ed.
- Maven build tool
- Git