Asgn5: towers AKA mm..OOMs
Dalal Arafeh

1. **In Part I, you implemented garbage collection—routines that clean up dynamically-allocated memory. How did you make sure the memory was all cleaned up? How did you check?**
    - list_destory prevents access to deallocated memory by ensuring a list is marked as empty by setting the header pointer to NULL after each node is freed.
        - valgrind is used to track memory allocation and deallocation. A successful cleanup is indicated if all the memory is freed and there is no memory leaks
        - Log ever node and hash table bucket deallocated with a unique ID
        - Verify that the list's head and hash table pointers are set to 'NULL' to ensure that the destroy function is correctly updating

2. **In Part II, you made a major optimization to the linked list optimization. What was it, and why do you think it changed the performance of bench1 so much?**
    - The major optimization to the linked list was changing the list structure to a hash table. Inserting each word from the dictionary with about 104K words took a long time. A hash table allows for storage and retrieval of key-value pairs by mapping keys to array indices using hash function.
    - It changed the performance of bench1 because hash tables speed up the overall operation by making the algorithm more efficient and scalable by allowing for insertions to take a consistent amount of time regardless of the size of the data structure.

3. **In Part III, you implemented hash tables. What happens to the performance of bench2 as you vary the number of buckets in your hash table? How many buckets did you ultimately choose to use?**
    - Increasing the number of buckets is like adding more floors to a tower, providing more room for people (hash table elements) to spread out across different levels. This expansion reduces congestion (collisions) within each floor (buckets), allowing for smoother mobility (faster access time). It also demands more space (memory) and higher costs (memory usage). On the other hand, decreasing the number of buckets is like reducing the number of floors from the tower, squeezing people into a smaller space. This saves memory, but can result in collisions and slower access time.
    - I have not chosen the number of buckets yet, but I could choose to narrow it down to prime numbers since it will greatly reduce collisions by ensuring I do not re-probe a previously probed location. This assignment also involves poor hashing so this would be helpful. The number of buckets would also need to balance the need for memory (space) with the efficiency of access times (mobility).

4. **How did you make sure that your code was free from bugs? What did you test, and how did you test it? In particular, how did you create inputs and check the output of uniqq?**
    - My uniqq.c could look like this

        Initialize hash table

Read lines from stdin
For each line, check if it is already in the hash table
If not, add it to the hash table
Output size of hash table (count of unique lines) after reading all lines

- I would need various text files to serve as input
  - An empty file
  - A file that contains only one line of text
  - A file that contains multiple unique files with no duplicates
  - A file that contains both duplicate and unique files
  - A file with lines that only differ by white spaces
  - A file with very long lines
  - A very large file
- Redirecting the test files into program using <
- To check the output of uniqq.c, I would use the command wc. This would compare the output of uniqq with the expected output. I could write a shell script that would automatically do it for me

Expectedoutput=$(test file using uniqq and wc commands)
Actualoutput=$(uniqq < testfile.txt)
If expectedoutput -eq to actualoutput then
Echo test passed
Else
Echo test failed
Fi

- I can use diff to test my expected output and actual output in text files
- Other tests
  - Make sure that items are inserted and stored correctly
  - When looking up an existing key that it returns the correct value
  - When looking up a non-existent key that it returns NULL or error
  - Make sure the program is correctly reading input and writing the output
  - Use valgrind to detect memory leaks
  - Binary files
  - Corrupt files (unusual character encodings)