/* Name: Megan Chu
* PID: A12814536
* Date: May 10, 2017
* CSE 100 PA3 C++ Huffman
* My compress/uncompress works on the small files checkkpoint1.txt and checkkpoint2.txt, but freezes/the
* code stops for some reason when testing the bigger test files warandpeace.txt and binary.dat
*/

Since the checkpoint submission I have fixed the error that gave seg fault in my build method for HCTree.cpp, and completed the previously incomplete uncompress.cpp and Checkpoint.pdf.

I attempted the part of the starpoint to try to make my compressed file as small as possible. And make it less than 50% the size of the hidden test case for the 0.5 bonus points.

1. I made code more eloquent by merging cases in the insert method where a new word was inserted without making a new path, and where a new word was inserted by making a new path into one case, this slightly improves the overall runtime.
2. I added a special case in the index method to return index 26 for a "space" character instead of resetting the index in the other methods every time there was a space character.
3. In predict completions I took the break statement in the while loop for the case where the number of words wanted by the user is greater than the number of possible words with the prefix, and added it to the while loop conditions instead for a cleaner looking code
4. For the insert method while iterating through every letter in the word, instead of having an if/else statement with two cases for if a path existed or it path didn't exist. I reduced it to one if statement that made a new node if the path didn't exist, and had a combined code executed every iteration for incrementing the iterator to the next letter in the word and traversing to the next corresponding node.
5. FOR PREDICT COMPLETIONS I cannot think of a more efficient runtime than what I currently have because although my code is very long, I think runtime-wise it is much more efficient.
   - I stored all the possible words AND their frequencies together in a vector so that I wouldn't have to use the find method to find the frequencies again when finding the words in order of highest frequency (which would cost more runtime).
   - The only thing I could think of was a priority queue to output the words with the highest frequency efficiently, and give me a cleaner looking code. HOWEVER, I did not implement this because I realized that if I had say 100 possible words with a prefix, and the user only wanted the top 10, I would have to run through the list of words 90 extra times to create the priority queue when I could have just ran through it 10 times without a priority queue. If list has n words, runtime is 100*n compared to current runtime of 10*n
   - I also thought of initially inserting words in order of frequency when they were found to the vector of all possible words. Although this would take less run time than sorting the vector after it was populated (stated in 5(b), new method would take runtime ~100*n/2), it would still take more time than finding individual highest frequency words if there are many possible words. Runtime would be 100*n/2 = 50*n compared to current runtime of 10*n.
   - I thought of another implementation that provided cleaner code, and compromised runtime less than the 2nd and 3rd bullet points under certain conditions. Upon the discovery of the first "num completion" possible words, they would be inserted into a vector, and the word with the smallest frequency in the vector would be kicked out once a new word with higher frequency was found. Instead of having to loop through the entire list of words every time to implement a priority queue, this method only requires looping through"num completions" number of words. However, one setback could be if num completions is greater than the total number of possible words, although it is highly unlikely that the user would demand such a big number of possible words returned. I still think my original method was more efficient because it does not have a "conditionally good" runtime, but a stable runtime.