Programming Assignment 4

CS 202 – Data Structures

Deadline: 21st April 2024, 11:55 PM

Lead TAs: Safa Salam, Saad Momin, Ahmed Faraz

Plagiarism Policy:

1. Students must not share the actual program code with other students.
2. Students must be prepared to explain any program code they submit.
3. Students cannot copy code from the Internet.
4. Students must indicate any assistance they received.
5. All submissions are subject to automated plagiarism detection.
6. Students are strongly advised that any act of plagiarism will be reported to the Disciplinary Committee

PART 1: HUFFMAN ENCODING (30 Marks)

In this part, you will be practically implementing the Huffman coding algorithm using your knowledge heaps in general. Huffman coding is a widely used algorithm for lossless data compression. It assigns variable-length codes to input characters, with shorter codes assigned to more frequent characters. Here are some key points to note:

1. Huffman coding uses a binary tree called the Huffman tree to encode characters.

2. Characters with higher frequencies are assigned shorter codes, while characters with lower frequencies are assigned longer codes.

3. Your task is to create a Huffman tree along with its frequency table, and to implement functions that can convert text input into its corresponding Huffman-encoded bitstrings, as well as decode Huffman-encoded bit strings back into the original text input.

You have been provided **huffman.h and huffman.cpp files**. Member function declarations of the huffman class have been provided in the huffman.h file. **You are not allowed to make any changes in the existing function declarations or the structure of the huffman and classes, otherwise, your test cases will not pass**. You must implement all the member functions declared in huffman.h. Additionally, **you are allowed to create as many helper functions as private members of the huffman and heap classes.**

- **You are prohibited from using the `vector.sort()` function to sort the vectors. Instead, you are encouraged to utilize your understanding of various sorting algorithms or any other suitable algorithms to facilitate sorting within your implementation, without relying on built-in functions to accomplish it for you.**
- **For the purposes of this assignment, you may assume that no nodes at any level of the Huffman tree have the same frequency. Therefore, there is no need to handle the scenario of identical nodes, as the sorting process can be randomized and may vary between students.**

Huffman Files:

Write the implementation for the following functions as described here:

Member Functions:

```
● string stringToBits(HuffNode* root, string input,
  unordered_map<char, string>& frequency_table)
```

The function takes your huffman tree's root, the input string and your frequency table as arguments. Your job is to compute the bit strings of every character and represent them within the frequency table as well as return the bitstring of the inputstring itself.

- `string bitsToString(HuffNode* root, int& i, string str)`

  Takes the root, the index of the string to decode, as well as the string as arguments. Your job is to traverse the bitstring and decode the bitstring back into the original message input string.

- `unordered_map<char, string> huffmanTree(string input);`

  Function to build the Huffman Tree and decode the given input text. This is where you essentially build your tree computing and adding nodes wherever necessary.

Note: *Please keep in mind that your implementation will be tested against hidden test cases that will not be disclosed to you. However, as long as your implementation is generic enough and you have used the correct approach while writing these functions, alongside ensuring that you have followed all the rules of the Huffman coding algorithm, your implementation should pass any and all hidden test cases and you do not need to worry about this at all.*

P.S *While the provided test cases may not explicitly evaluate the level-order traversal of your tree, it's important to note that your implementation will be assessed using hidden test cases. These tests will also verify the level-order traversal of the tree you construct, assuming there are no ties. Consequently, the level-order traversal should yield consistent results for all implementations.*

You are welcome to employ as many helper functions as necessary for your implementation. ***However, refrain from importing any external libraries for this purpose.***

PART 2: EXTERNAL SORT (30 Marks)

In this part, you will be practically implementing the external sorting algorithm. According to Wikipedia, external sorting is a technique used to sort large datasets that do not fit entirely in memory. It involves dividing the dataset into smaller chunks, sorting each chunk in memory, and then merging the sorted chunks into a single sorted file.

You have been provided with the following files: ExternalSort.h and ExternalSort.cpp. The member function declarations of the ExternalSort class as well as the private member variables have been provided in the ExternalSort.h file. You are not allowed to make any changes in the existing function declarations or the member variable declarations; otherwise, your test cases will not pass. You must implement all the member functions declared in ExternalSort.h. Additionally, you are allowed to create as many helper functions as private members of the ExternalSort class.

MEMBER FUNCTIONS:

Write the implementation for the following functions as described here:

- void sort():

  Sorts the input file using the external sorting algorithm.
  Divides the input file into smaller chunks, sorts each chunk, and merges the sorted chunks into a single sorted file. Ensure to use the most efficient method available.

- void deleteTempFiles():

  Deletes all temporary files created during the sorting process.

- void mergeSortedChunks():

  Merges the sorted chunks into the final output file. Make sure to implement this process efficiently. You can call this function within your sort function.

HELPER FUNCTIONS:

While it is highly encouraged to think of helper functions yourself, we would strongly suggest you to make the following helper function before making any of the above functions as it will remain useful to you while implementing several of the above functions:

void mergeChunks()

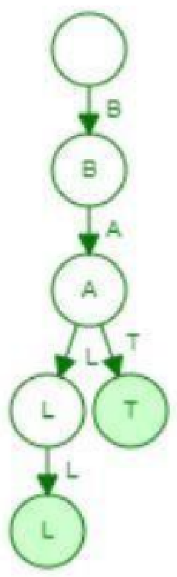
Merges two sorted chunks into a single sorted chunk.
This function will be useful in the mergeSortedChunks() function.

Note: Y*our implementation will be tested against hidden test cases that will not be disclosed to you. However, as long as your implementation is generic enough and you have used the correct approach while writing these functions, alongside ensuring that you have followed all the rules of standard external sorting, your implementation should pass any and all hidden test cases.*

PART 3: TRIES (40 Marks)

In this part, you will be practically implementing the trie data structure. According to Wikipedia, a trie is a type of search tree, a tree data structure used for locating specific keys from within a set. These keys are most often strings, with links between nodes defined not by the entire key, but by individual characters. In our case, we will be using words from the English language as keys and Alphabets of the English language as individual characters. Each node will be storing a character and a vector of nodes as its child nodes. For example, a trie with keys bat and ball will look like this: There are three really important things to note here:

1. At each level, characters are stored in an alphabetical order e.g., at Level 3, bothL and T are children of A but L must appear before T which is why the word ball is stored before the word bat even if the word bat is inserted before the word ball. This is just how a standard trie works



2. For two keys having n same starting characters (where n is an integer > 1), the first n characters are not duplicated while inserting the second key within the trie. For example, if ball is inserted as the second key within the trie while bat already exists, the characters ‘b’ and ‘a’ are not duplicated. Instead, it creates a new branch from a towards the left and stores the first ‘l’ within the ball as a child of a towards the left of ‘t’ from bat.

3. While the given image shows only capital letters within the trie, your implementation should work on both capital and small letters. However, you do not need to handle this separately. As long as your functions have been implemented correctly, they will be general enough to handle all sorts of characters (even non-English characters). So, you do not need to worry about this.

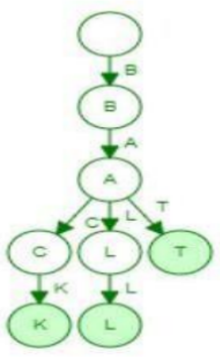
You have been provided trie.hpp and trie.cpp files. Member function declarations of the trie class as well as the structure for a Node have been provided in the trie.hpp file. You are not allowed to make any changes in the existing function declarations or the node structure otherwise your test cases will not pass. You must implement all the member functions declared in trie.hpp. Additionally, you are allowed to create as many helper functions as private members of the trie class.

MEMBER FUNCTIONS:

Write the implementation for the following functions as described here:

void insertWord(string word)

● Inserts a word from the English Language into the trie
● You must make sure that it follows the rules mentioned above after it has been successfully implemented.
● For example, when insertWord(“back”) is called on the trie shown above, the resultant trie will look like this:



<u>bool search(string word)</u>

● Searches for a word within the trie. Returns true if a word is found within the trie, otherwise returns false
● For example, search("Ball") returns true whereas search("Base") returns false string

<u>longestSubstr(string word)</u>

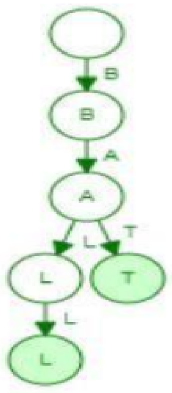
● Returns the first n characters of the provided argument as a string after searching for the word within the trie. If not, even a single character is found within the trie, it returns an empty string.
● For example, longestSubstr("Base") returns "Ba" whereas longestSubstr("random") returns "".
Vector

<u>getTrie()</u>

● Traverses the entire trie and returns all the words present within the trie, in alphabetical order, in a vector
● For example, applying this function on the above trie will return a vector with the following strings: "Back", "Ball", "Bat"

<u>void deleteWord(string word)</u>

● Removes the word provided as an argument to it, from the trie. However, it must ensure not to delete the characters which are common ancestors to several words.
● For example, when deleteWord("Back") is called on the above trie, the resultant trie will look like this:



● As you can see, the characters ‘B’ and ‘A’ did not get deleted when deleteWord(“Back”) was called because they were ancestors to the words “Ball” and “Bat”.

HELPER FUNCTIONS

While it is highly encouraged to think of helper functions yourself, we would strongly suggest you to make the following helper function before making any of the above functions as it will remain useful to you while implementing several of the above functions:
findChar() Make a function that searches if a particular character exists within the trie at one particular level. When successfully implemented, this function should tell you if that character is found or not. The return type and the parameters of this function are totally up to you to decide.

TRIE VISUALIZATION

The following link is being provided to you to better help you out with visualizing tries. We would recommend you to visit this link and try out different trie operations such as insertion and deletion and then see how the trie looks like as a result of these operations. This will help you better understand what's expected from you in this part.

https://www.cs.usfca.edu/~galles/visualization/Trie.html

Note: *Please keep in mind that your implementation will be tested against hidden test cases that will not be disclosed to you. However, as long as your implementation is generic enough and you have used the correct approach while writing these functions, alongside ensuring that you have followed all the rules of a standard trie, your implementation should pass any and all hidden test cases and you do not need to worry about this at all.*

TESTING:

To test the implementation of your code, compile and run the test files using the following commands :

TASK 1:

TASK 2:

```
g++ ExternalSort.cpp TestExternalSort.cpp -o a
./a
```

TASK 3: (in WSL)

```
g++ test_tries.cpp -pthread -std=c++11
./a.out
```

Submission guidelines:

You only have to submit .cpp files for both tasks. If you add any helper function in the header files then you may submit those as well.

Zip the complete folder and use the following naming convention:
PA4_<roll number>.zip

For example, if your roll number is 25100018 then your zip file name should:
PA4_25100018.zip

All submissions must be uploaded on LMS before the deadline.
You are allowed 5 "free" late days during the semester (that can be applied to one or more assignments; the final assignment will be due tentatively on

the final day of classes, i.e., before the dead week and cannot be turned in late. The last day to do any late submission is also the final day of classes, even if you have free late days remaining).

If you submit your work late for any assignment once your 5 “free” late days are used, the following penalty will be applied:

- 10% for work submitted up to 24 hours late
- 20% for work submitted up to 2 days late
- 30% for work submitted up to 3 days late
- 100% for work submitted after 3 days (i.e., you cannot submit assignments more than 3 days late after you have used your 5 free late days.