CS 202, Summer 2023

Homework 2 – Trees

Due: 23:59, July 25, 2023

Important Notes

Please do not start the assignment before reading these notes.

1. Before 23:59, July 25, upload your solutions in a single **ZIP** archive using Moodle submission form. Name the file as studentID_secNo_hw2.zip.
2. Your ZIP archive should contain the following files:
   - **`hw2.pdf`**, the file containing the answers to Questions 1, 2 and 4, and the sample output of the program.
   - **NgramTree.cpp, NgramTree.h, and hw2.cpp,** and any additional files if you wrote additional classes in your solution, and the **`Makefile`**.
   - Do not forget to put your name and student id in all of these files. Well comment your implementation. Add a header as given below to the beginning of each file:

```
/*
 * Title: Trees
 * Author: Name Surname
 * ID: 21000000
 * Section: 1
 * Assignment: 2
 * Description: description of your code
 */
```

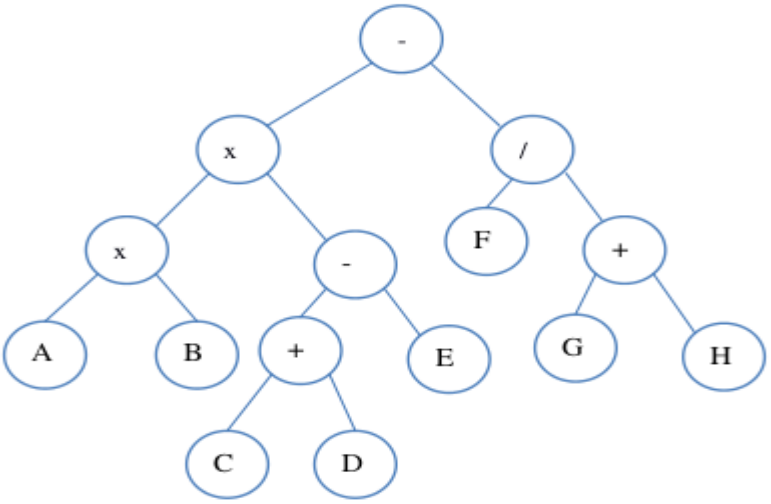
   - Do not put any unnecessary files such as the auxiliary files generated from your favorite IDE. Be careful to avoid using any OS dependent utilities (for example to measure the time).
   - You should prepare and upload **handwritten** answers for Question 1, 2 and 4 (in other words, do not submit answers prepared using a word processor).
   - Use the exact algorithms shown in lectures.
   - Keep all the files before you receive your grade.
3. Although you may use any platform or any operating system to implement your algorithms and obtain your experimental results, your code should work on the `dijkstra` server (dijkstra.ug.bcc.bilkent.edu.tr). We will compile and test your programs on that server. Thus, you will lose a significant amount of points if your C++ code does not compile or execute on the `dijkstra` server.
4. This homework will be graded by **your TA, Saeid Karimi (saeed.karimi at bilkent edu tr)**. Thus, you may ask your homework related questions directly to him.

Attention: For this assignment, you are allowed to use the codes given in our textbook and/or our lecture slides. However, you ARE NOT ALLOWED to use any codes from other sources (including the codes given in other textbooks, found on the Internet, belonging to your classmates, etc.). Furthermore, you ARE NOT ALLOWED to use any data structure or algorithm related function from the C++ standard template library (STL).

Do not forget that plagiarism and cheating will be heavily punished. Please do the homework yourself.

Question 1 (10 points)

Give the prefix, infix, and postfix expressions obtained by preorder, inorder, and postorder traversals, respectively, for the expression tree below:



Question 2 (10 points)

Draw the initially empty Binary Search Tree after operations as follows (show all intermediate steps):

insert 43, 28, 32, 20, 90, 83, 101, 84, 23, 76, 53, 13, 73, 91; then delete 53, 23, 43.

Question 3, Programming Assignment (60 points)

You are to write a C++ program to count the frequency (number of occurrences) of n-grams in a text file. Definition of n-gram is simple: it is the number of consecutive letters in a given text. For example, for the word bilkent the 2-grams (bigrams) are bi, il, lk, ke, en, nt. You may ignore any capitalizations and assume that the text file contains only English letters 'a'...'z', 'A'…'Z', and the blank space to separate words. Your program should take the value of *n* as a parameter and construct the corresponding BST accordingly. While processing the input text, if your program encounters a word that has length smaller than the value of parameter *n,* you can simply ignore that word and process following words.

You are to use a pointer based implementation of a **Binary Search Tree** (BST) to store the n-grams and their counts. (You can use the source codes available in the course book as well as you can implement a BST yourself.) Each node object is to maintain the associated n-gram as a string, its current count as an integer, and left and right child pointers. On top of the regular operations that a BST has, you must implement the following functions:

- **`addNgram:`** adds the specified n-gram in the BST if not already there; otherwise, it simply increments its count.
- **`generateTree:`** reads the input text and generates a BST of n-grams. In this function, you should detect all of the n-grams in the input text and add them to the tree by using the **`addNgram`** function. This function also requires the parameter *n*.

- **`getTotalNgramCount`**: recursively computes and returns the total number of n-grams currently stored in the tree.
- **`isComplete`**: computes and returns whether or not the current tree is a complete tree.
- **`isFull`**: computes and returns whether or not the current tree is a full tree.

- **`operator<<`**: recursively prints each n-gram in the tree in alphabetical order along with their frequencies. This should be a global function.

Below is the interface of an NgramTree class for implementing the above functionality as well as a `main` function to test it with a sample input text file. These will be used for evaluation purposes. Make sure your code runs correctly against these. We will test your program extensively.

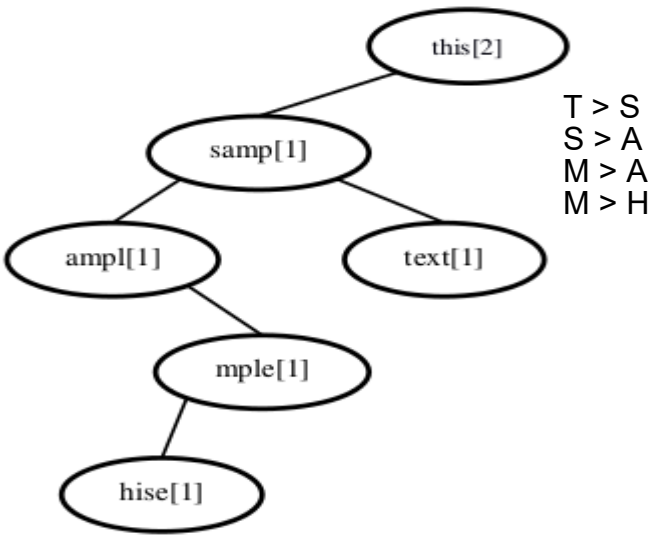
```
class NgramTree {
public:
    NgramTree();
    ~NgramTree();

    void addNgram( const string& ngram );
    int  getTotalNgramCount() const;
    bool isComplete() const;
    bool isFull() const;
    void generateTree( const string& fileName, const int n );

private:
    // ...

    friend ostream& operator<<( ostream& out, const NgramTree& tree );
};
```

```
// hw2.cpp
#include <iostream>
#include <string>

using namespace std;

#include "NgramTree.h"
```

```
int main( int argc, char** argv ) {
    NgramTree tree;
    string fileName( argv[1] );
    int n = atoi( argv[2] );
    tree.generateTree( fileName, n );

    cout << "\nTotal " << n << "-gram count: " << tree.getTotalNgramCount() <<
endl;
    cout << tree << endl;

    cout << n << "-gram tree is complete: " << (tree.isComplete() ? "Yes" : "N
<< endl;

    // Before insertion of new n-grams
   cout << "\nTotal " << n << "-gram count: " << tree.getTotalNgramCount() <<
endl;

   tree.addNgram( "samp" );
   tree.addNgram( "samp" );
   tree.addNgram( "zinc" );
   tree.addNgram( "aatt" );

   cout << "\nTotal " << n << "-gram count: " << tree.getTotalNgramCount() <<
endl;
   cout << tree << endl;
   cout << n << "-gram tree is complete: " << (tree.isComplete() ? "Yes" : "No
<< endl;

   cout << n << "-gram tree is full: " << (tree.isFull() ? "Yes" : "No") << en

   return 0;
}
```

```
// input.txt
this is sample text
and thise is all

// Sample output
Total 4-gram count: 6
"ampl" appears 1 time(s)
"hise" appears 1 time(s)
"mple" appears 1 time(s)
"samp" appears 1 time(s)
"text" appears 1 time(s)
"this" appears 2 time(s)

4-gram tree is complete: No
4-gram tree is full: No

Total 4-gram count: 8
"aatt" appears 1 time(s)
"ampl" appears 1 time(s)
"hise" appears 1 time(s)
"mple" appears 1 time(s)
"samp" appears 3 time(s)
"text" appears 1 time(s)
"this" appears 2 time(s)
"zinc" appears 1 time(s)

4-gram tree is complete: No
4-gram tree is full: No
```

The following is the BST constructed for the input text whe $n = 4$.



Question 4 (20 points)

Analyze the worst-case running time complexities of the **addNgram** and **operator<<** functions in the previous question using the big-O notation.