CS 202, Fall 2019
Homework #2 – Binary Search Trees
Due Date: November 09, 2019

Important Notes

Please do not start the assignment before reading these notes.

- Before 23:55, November 9, upload your solutions in a single **ZIP** archive using Moodle submission form. Name the file as `studentID_hw2.zip`.
- Your ZIP archive should contain the following files:
  - `hw2.pdf`, the file containing the answers to Question 1 and 3,
  - `NgramTree.h`, `NgramTree.cpp`, `main.cpp` files which contain the C++ source codes, and the `Makefile`.
  - Do not forget to put your name, student ID, and section number in all of these files. Well comment your implementation. Add a header as in Listing 1 to the beginning of each file:

Listing 1: Header style

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

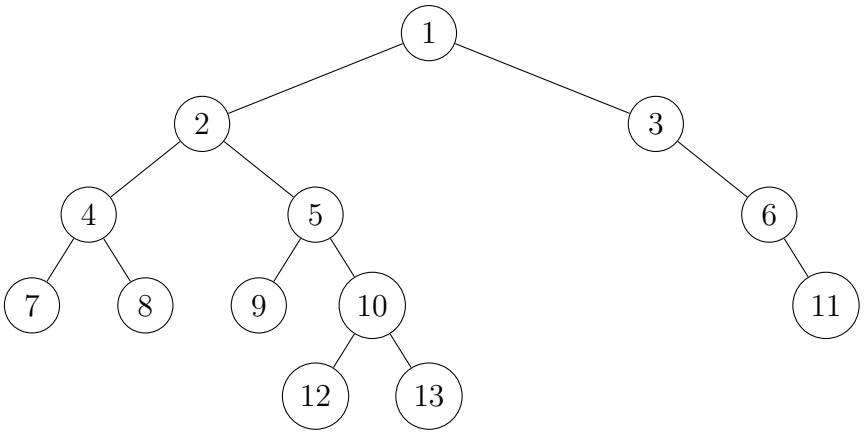
  - Do not put any unnecessary files such as the auxiliary files generated from your favorite IDE. Be careful to avoid using any OS dependent utilities (for example to measure the time).

- **You should prepare the answers of Question 1 and 3 using a word processor (in other words, do not submit images of handwritten answers).**
- Please use the algorithms as exactly shown in lectures.
- Although you may use any platform or any operating system to implement your algorithms and obtain your experimental results, your code should work in a Linux environment (specifically on the **Dijkstra** 'dijkstra.ug.bcc.bilkent.edu.tr' server). We will compile your programs with the g++ compiler and test your codes in a Linux environment. Thus, you may lose significant amount of points if your C++ code does not compile or execute in a Linux environment.
- Please make sure that you are aware of the homework grading policy that is explained in the rubric for homeworks
- This homework will be graded by your TA, Alihan Okka. Thus, please contact him directly for any homework related questions.

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 – 20 points

(a) [*10 points*] What are the preorder, inorder, and postorder traversals of the binary tree below:



(b) [*10 points*] Insert 50, 40, 80, 30, 45, 70, 90, 10, 60, 75, 85 to an empty Binary Search Tree. **Show only the final tree after all insertions.** Then delete 45, 40, 80, 75, 50 in given order. **Show only the final tree after all deletion operations.** Use the exact algorithms shown in lectures.

Question 2 – 70 points

Given a text file, you are to write a program to count the frequency of n-grams which is a concept found in Natural Language Processing. Definition of n-gram is simple: it is the number of consecutive letters in a given text. For example, the 2-grams for the word bilkent are bi, il, lk, ke, en, nt. Your program should take the value of n as a parameter and construct the corresponding Binary Search Tree (BST) accordingly.

You are to use a pointer based implementation of a BST to store the n-grams and their counts. (You can use the source codes available in the course book or slides 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 it is 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.
- **printNgramFrequencies:** recursively prints each n-gram in the tree in alphabetical order along with their frequencies.
- **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.

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. 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.

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.

Question 3 – 10 points

Analyze the worst-case running time complexities of the **addNgram** and **printNgramFrequencies** functions in the previous question using the Θ (Big-Theta) notation. Argue your case.

Listing 2: NgramTree.h

```
#include <string>

class NgramTree{
    public:
        NgramTree();
        ~NgramTree();
        void addNgram(std::string ngram);
        int getTotalNgramCount();
        void printNgramFrequencies();
        bool isComplete();
        bool isFull();
        void generateTree(std::string fileName, int n);
        ...
    private:
        ...
};
```

Listing 3: main.cpp

```
#include "NgramTree.h"
#include <iostream>
#include <string>

using namespace std;

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

    tree.generateTree(fileName, n);
    cout << "\n Total " << n << "-gram count: "
        << tree.getTotalNgramCount() << endl;
    tree.printNgramFrequencies();
    cout << n << "-gram tree is complete: "
        << (tree.isComplete() ? "Yes" : "No") << endl;

    //Before insertion of new n-grams
    cout << "\n Total " << n << "-gram count: "
        << tree.getTotalNgramCount() << endl;
    tree.addNgram("samp");
    tree.addNgram("samp");
    tree.addNgram("zinc");
    tree.addNgram("aatt");
    cout << "\n Total " << n << "-gram count: "
        << tree.getTotalNgramCount() << endl;
    tree.printNgramFrequencies();
    cout << n << "-gram tree is complete: "
        << (tree.isComplete() ? "Yes" : "No") << endl;
    cout << n << "-gram tree is full: "
        << (tree.isFull() ? "Yes" : "No") << endl;
    return 0;
}
```

Listing 4: Sample Input-Output

```
// Sample input
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 when n=4.