CS 202 - Spring 2022
Homework 2 - Trees
Due: 23:55, March 25, 2022

Important Notes

Please do not start the assignment before reading these notes.

1. Before 23:55, March 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 and 3.
   - `BinaryNode.h`, `BinaryNode.cpp`, `BinarySearchTree.h`, `BinarySearchTree.cpp`, `analyze.cpp`, and `bst-analysis.txt` files as well as any additional class and its header 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 given below to the beginning of each file:

```
/*
 * Title: Trees
 * Author: Name Surname
 * ID: 22000000
 * Section: 9
 * 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 upload handwritten answers for Question 1 (in other words, do not submit answers prepared using a word processor).
   - Use the exact algorithms shown in lectures.

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 significant points if your C++ code does not compile or execute on the `dijkstra` server.

4. This homework will be graded by your TA, Navid Ghamari. Thus, please contact directly (navid.ghamari at bilkent.edu.tr) for any homework-related questions.

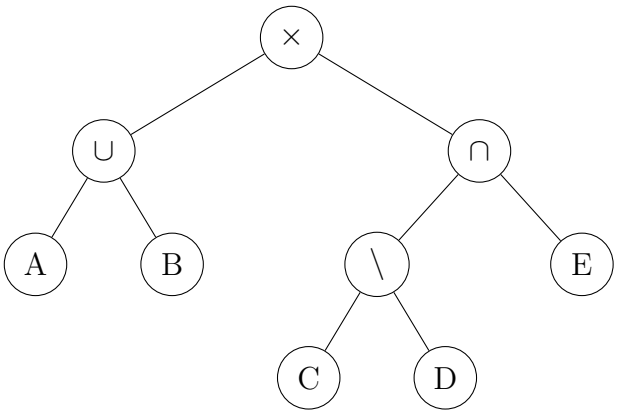
Attention: For this assignment, you are allowed to use the codes given in our textbook and/or our lecture slides. However, you ARE NOT ALLOWED to use any codes from other sources (including the codes given in other textbooks, found on the Internet, belonging to

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



(b) (9 points) Insert `50`, `20`, `80`, `10`, `65`, `75`, `45`, `90`, `70`, `60`, `30`, `40`, `63` to an empty Binary Search Tree, and then delete `10`, `75`, `80`, `20`, `50` in the given order. Show the evolution of the BST after each insertion and deletion operation.

(c) (5 points) A binary search tree has a postorder traversal of `3`, `5`, `11`, `17`, `13`, `7`, `2`, `23`, `37`, `43`, `41`, `31`, `53`, `47`, `29`, `19`. Give the corresponding binary search tree. What is its preorder traversal?

Question 2 (65 points)

You will write a pointer-based implementation of the Binary Search Tree (BST) with various functions.

The Binary Search Tree shall keep integer data values at the nodes. Each node of the tree shall keep the corresponding integer data value and left and right child pointers. In addition to those standard attributes, the nodes shall also keep a *size* attribute and a *parent pointer* attribute. Ensure that all of your operations on the tree also maintain the size and the parent pointer attributes properly such that they satisfy the following:

- The size attribute keeps the number of nodes in the subtree referred by this node.
- The parent pointer points to the parent node, if any, otherwise it is `nullptr`.

Your node implementation should be named `BinaryNode`. The header and implementation should reside in `BinaryNode.h` and `BinaryNode.cpp` files, respectively.

Your tree implementation should be named `BinarySearchTree`. The header and implementation should be in `BinarySearchTree.h` and `BinarySearchTree.cpp` files, respectively.

The `BinarySearchTree` class should provide the following public methods.

`bool` `isEmpty()`: Returns true if BST is empty; otherwise false.

`int` `getHeight()`: Returns the height of the BST.

`int` `getNumberOfNodes()`: Returns the number of nodes in the BST.

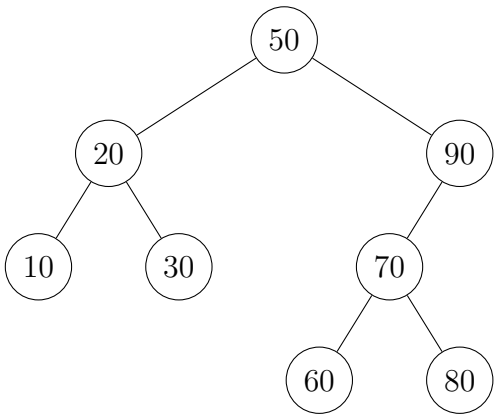


Figure 1: An example Binary Search Tree

bool `add(int newEntry)`: Adds a new node containing a given data item to the BST. Returns false and does not add if `newEntry` is less than or equal to zero or `newEntry` already exists in the current BST. Returns true if the addition is successful.

bool `remove(int anEntry)`: Removes the node containing the given data item from BST. True if the removal is successful, or false if not.

bool `contains(int anEntry)`: Tests whether the given data item occurs in the BST. True if the BST contains the given data item, or false if not.

void `inorderTraverse()`: Traverses the BST in inorder and prints the data of each node in the traversal order.

int `getWidth()`: Returns the width of the tree. The *width* is defined as the number of nodes at the most populated depth.

For instance, in the tree of Figure 1,

- number of nodes in the first depth (root) is 1
- number of nodes in the second depth is 2
- number of nodes in the third depth is 3
- number of nodes in the fourth depth is 2

The most populated depth is the third, and the number of nodes at this depth is 3. So, the width of this tree is 3.

int `count(int a, int b)`: Returns the number of nodes in the BST with data values within a specific range. In other words, it is the number of nodes with the data value greater than or equal to a, and less than or equal to b. You should implement an efficient function – that is, your function should not traverse any parts of the tree that it does not need to traverse.

For example, in the tree of Figure 1,

- `count(20, 60)` should return `4`
- `count(10, 30)` should return `3`
- `count(35, 45)` should return `0`

int `select(int anEntry)`: Returns the `0`-based index of `anEntry` in the sorted sequence of the items in the BST. If `anEntry` is not in the BST, then return `-1`. You

should implement an efficient function – that is, your function should not traverse any parts of the tree that it does not need to traverse.

For example, in the tree of Figure 1,

- `select(10)` should return `0`
- `select(50)` should return `3`
- `select(70)` should return `5`
- `select(40)` should return `-1`

`int` `successor(`**`int`** `anEntry)`: Returns the inorder successor of `anEntry`. If `anEntry` does not exist in the BST, returns `-1`. If `anEntry` is the last entry in the BST, returns `-2`.

For example, in the tree of Figure 1,

- `successor(10)` should return `20`
- `successor(50)` should return `60`
- `successor(70)` should return `80`
- `successor(40)` should return `-1`
- `successor(90)` should return `-2`

You are free to write helper functions to accomplish the tasks required from the above functions. Use the given file names and function signatures during implementation. You will not submit a `main.cpp` file, instead we will use our `main.cpp` file during evaluation. Therefore, it is important to use given file names and function signatures.

Question 3 (15 points)

In this question, you will analyze how the height of BST behaves on inserts/deletes of random sequence of numbers.

Write a global function, `analyzeBST()` in a file named `analyze.cpp`. This function shall do:

- Create an array `arr` of `10000` of random integers.
- Create an empty BST and insert the array elements to the BST. At each `100` iterations, output the height of the tree.
- Shuffle `arr`.
- Delete the numbers on `arr` from the BST, output the height of the tree at each `100` iterations.

Your output should be in the following format:

```
Random BST size vs. height (Insertion)
---------------------------------------
100 13
199 14
297 15

Random BST size vs. height (Deletion)
---------------------------------------
8528 30
8428 30
...
0 0
```

Note that, in the output, the number of nodes in BST may not be a multiple of 100, due to possible duplications in `arr`.

Run this method, and save the output on a file `bst-analysis.txt`.

Plot different plots of insertion series and deletion series on the same chart. Chart should have the number of elements of the tree on x-axis and height of the tree on y-axis. You may use log-scale to present the data better. Discuss whether the result matches your expectations.