CS 202, Fall 2022
Homework 3 - Balanced Search Trees, Heaps

Due: 23:55, December 5, 2022

Important Notes

Please do not start the assignment before reading these notes.

- Before 23:55, December 5, upload your solutions in a single **ZIP** archive in Moodle. Name the file as `studentID_hw3.zip`.
- Your ZIP archive should contain the following files:
  - Your handwritten answers to questions 1 and 2 in a file named `firstname_lastname_hw3.pdf`.
  - Your code files for question 3. The names of your code files must be `DictionarySearchTree.h`, `DictionaryBST.h`, `DictionaryAVLTree.h`, `Dictionary23Tree.h`, `DictionarySearchTree.cpp`, `DictionaryBST.cpp`, `DictionaryAVLTree.cpp`, and `Dictionary23Tree.cpp`. You can also write additional files but do not include your test `main` function in any of these files.
  - Your `Makefile` which compiles all your code and creates an executable file named `hw3`. Check out this tutorial for writing a simple make file: tutorial.
- Add a header as in Listing 1 to the beginning of each file:

Listing 1: Header style

```
/**
 * Title: AVL Trees
 * Author: Name Surname
 * ID: 21000000
 * Section: 1
 * Assignment: 3
 * Description: description of your code
 */
```

- Do not forget to write comments at important parts 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.
- You should prepare and upload handwritten answers for question 1 and 2 (in other words, do not submit answers prepared using a word processor).
- In question 3, you **MUST** follow **the design requirements** about polymorphism and function overloading discussed below.
- Although you may use any platform or any operating system to implement your algorithms, 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 may lose a significant amount of points if your C++ code does not compile or execute on the dijkstra server.
- Any violation of these may cause a significant loss from your grade.
- This homework will be graded by your TA, Hakan Sivük. Thus, please contact him directly (hakan.sivuk at bilkent edu tr) for any homework related questions.

Attention: For this assignment, you must use your own implementations of binary search trees, AVL trees, and 2-3 trees. You can adapt the codes and pseudocodes in the Carrano book. However, you are NOT allowed to use any existing codes from other sources (including the codes given in other textbooks, found on the internet, and belonging to your classmates, etc.). Furthermore, you are NOT allowed to use any data structure or 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 (30 points)

Assume that we have the following balanced-searched tree implementations:

a) AVL tree

b) 2-3 tree

c) 2-3-4 tree

Starting with an empty balanced search tree, we would like to insert the following keys into the tree in the given order:

```
30 45 60 16 22 36 10 25
```

and then, delete the keys `10 45 22` (in the given order) from the tree. Note: while deleting an internal node, its inorder successor should be used as the substitute if needed.

Show the underlying tree for the AVL, 2-3, and 2-3-4 implementations after *each* insertion and deletion by using the *exact algorithms* that we discussed in the lectures.

Question 2 (10 points)

Suppose we wish to create a binary heap containing the keys

```
G J D T U C B Q B X C Z. (All comparisons use alphabetical order.)
```

a) Show the resulting max-heap if we build it using successive `heapInsert()` (see page 43 of slides on Heaps) operations (starting from F).

b) Show the resulting max-heap if we build it using the algorithm for building a heap from an array (that successively calls `heapRebuild()`, see page 53 of slides on Heaps).

Question 3 (60 points) (Programming Assignment)

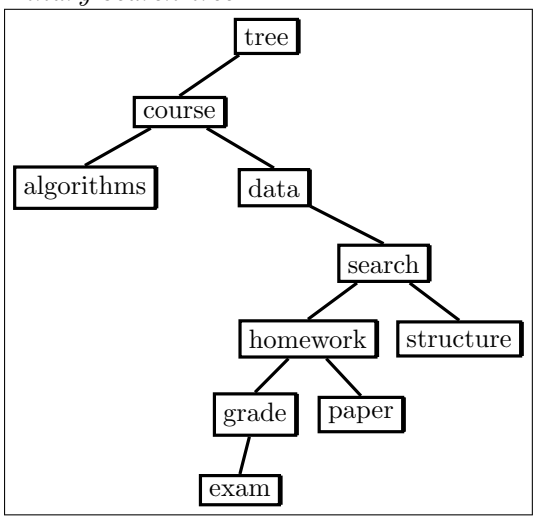
In this question, you are asked to write a C++ program for querying a dictionary of English words. The queries will be run on a binary search tree-based (not necessarily balanced), an AVL tree-based, and a 2-3 tree-based implementation of the dictionary. Your program will first read the contents of the dictionary from an input file, and build the search trees. The file will contain an unsorted list of words with each word given in a separate line. Your program MUST build the trees by inserting these words in the order that they appear in the file.

For example, for the given dictionary, the following trees are constructed. You can assume that all words are unique.

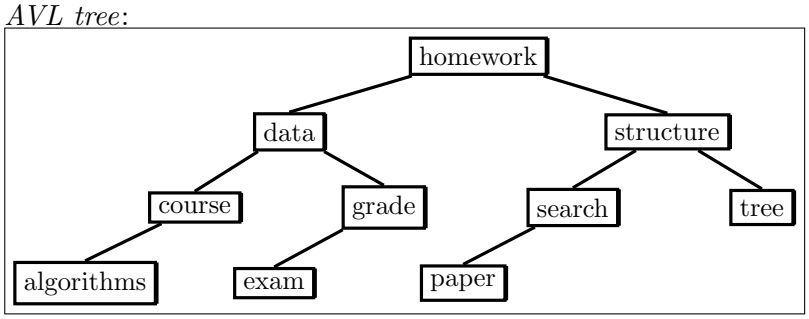
Dictionary:

tree
course
data
search
structure
algorithms
homework
grade
exam
paper

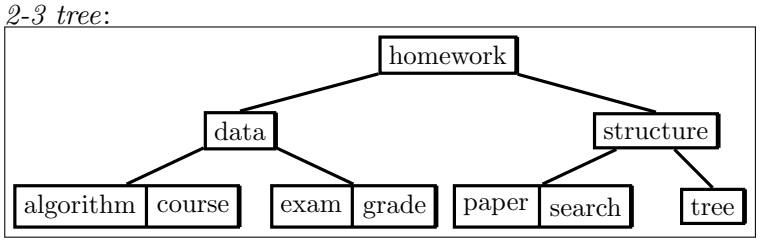
Binary search tree:



AVL tree:



2-3 tree:



After building the trees, your program will read another input file that contains the query words (each word in a separate line), search each of these words in the dictionary, and generate an output file consisting of the search statistics. Each line in the output file must present the search result for the corresponding word in the query file. The line should show the word, whether it is found in the dictionary or not (1 or 0), and the number of key (string) comparisons made during the search. You should also show the maximum and average number of comparisons made.

For example, for the given query words below, the following results are obtained. You can assume that the comparisons are case insensitive.

Queries:

information
algorithms
homework
word
search
paper
binary
exam

Binary search tree:

information	0	6
algorithms	1	3
homework	1	5
word	0	1
search	1	4
paper	1	6
binary	0	3
exam	1	7
# of queries: 8		
Maximum # of comparisons: 7		
Average # of comparisons: 4.3750		

AVL tree:

information	0	4
algorithms	1	4
homework	1	1
word	0	3
search	1	3
paper	1	4
binary	0	4
exam	1	4

# of queries: 8
Maximum # of comparisons: 4
Average # of comparisons: 3.3750

2-3 tree:

information	0	3
algorithms	1	3
homework	1	1
word	0	3
search	1	4
paper	1	3
binary	0	4
exam	1	3

# of queries: 8
Maximum # of comparisons: 4
Average # of comparisons: 3.0000

Use the following interface for implementing the dictionary using search trees. **An important design goal in this assignment is to avoid code duplication where you should factor out and put all common functionality into a base class named `DictionarySearchTree`, and make the other three classes extend it, reuse the functions in the base class, and override only the needed functions.** Thus, you are expected to study how polymorphism and virtual functions are used in C++ when you design and formulate your solution in this assignment.

```
//Search tree implementation for the dictionary
class DictionarySearchTree {
    public:
        //Constructor that constructs the dictionary from the input file
        DictionarySearchTree( string dictionaryFile );

        //Destructor
        virtual ~DictionarySearchTree();

        //Inserts the given word into the dictionary
        virtual void insert( string word );

        //Searches the given word in the dictionary, and returns the number
        //of comparisons made and whether the word is found or not
        virtual void search( string word, int& numComparisons, bool& found ) const;

        //Searches all words in the query file in the dictionary, and
        //summarizes the results in the output file
        virtual void search( string queryFile, string outputFile ) const;
        ...
};
```

Following the interface for the class named `DictionarySearchTree`, you MUST design and implement three additional classes: `DictionaryBST` for a binary search tree-based implementation, `DictionaryAVLTree` for an AVL tree-based implementation, and `Dictionary23Tree` for a 2-3 tree-based implementation. The class definitions must be in the files named `DictionarySearchTree.h`, `DictionaryBST.h`, `DictionaryAVLTree.h` and `Dictionary23Tree.h`, and the class implementations must be in the files `DictionarySearchTree.cpp`, `DictionaryBST.cpp`, `DictionaryAVLTree.cpp` and `Dictionary23Tree.cpp`. You can use additional files in your implementation. You can also define additional member functions and data members but do not modify the definitions of the member functions given for `DictionarySearchTree` above.

An example `main` function for using these classes is also given. You are expected to test your code by writing a driver `main` function but do not submit your `main` function as part of your code.

```
int main() {

    DictionarySearchTree* myTrees[ 3 ];

    myTrees[ 0 ] = new DictionaryBST( "dictionary.txt" );
    myTrees[ 1 ] = new DictionaryAVLTree( "dictionary.txt" );
    myTrees[ 2 ] = new Dictionary23Tree( "dictionary.txt" );

    string outFiles[ 3 ] = { "outBST.txt", "outAVLTree.txt", "out23Tree.txt" };

    int i;
    for ( i = 0; i < 3; i++ ) {
        myTrees[ i ]->search( "query.txt", outFiles[ i ] );
    }

    for ( i = 0; i < 3; i++ ) {
        delete myTrees[ i ];
    }

    return 0;
}
```