Middle East Technical University　　　　Department of Computer Engineering

CENG 213

Data Structures

Spring 2018-2019

Programming Assignment 2

Due date: 15 April 2019, Monday, 23:55

1 Objectives

In this assignment, you are expected to implement a book index, which helps you to find page number of a term in a book.

In an index, the terms are divided into groups according to their first letter. These letters are ordered in alphabetical order. The terms in each letter group are in lexicographical order among themselves, as well. Your objective is simply to group terms whose first letter are the same and list terms in each group with respect to lexicographical ordering. To achieve this, you are expected to implement some methods related to some functionalities of the index. An index example is provided below.

```
A:
AA-trees 1-5
Abstract data types 12 15
Algorithm design 13
D:
DAGs 45 89 134
N:
Nested loops 56 123-126
New line character 22-25 68 207
```

Each line beginning with a term has the page numbers of the occurrences of that term in the book. Note that page numbers can be a single number as well as an interval. For instance, “Nested loops” occurs in page 56 and also from page 123 to 126. In Figure 1, the binary search tree representation of above index is illustrated. Please note that, the first letters of terms are arranged as binary search tree and each node for a letter also holds a separate binary search tree to store terms. The nodes in the binary search tree are ordered alphabetically. In the following description of the homework, the overall binary search tree in Figure 1 is referred to as Index; the nodes of this tree (which includes letters) are referred to as Index Node; the secondary tree stored in each node are referred to as TermTree; the nodes of the secondary tree as TermTreeNode.

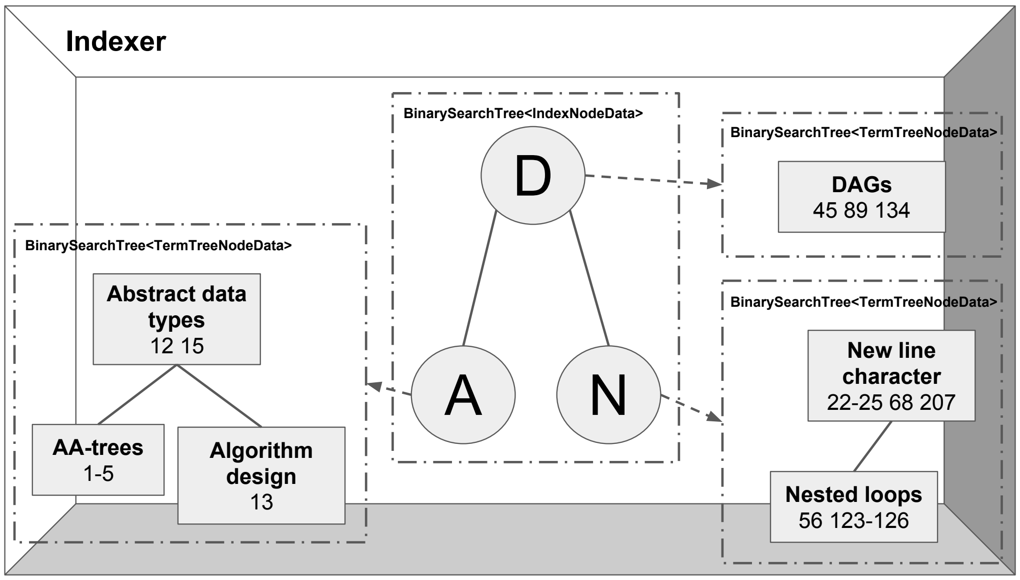
Indexer
BinarySearchTree<IndexNodeData>
D
A
N
BinarySearchTree<TermTreeNodeData>
Abstract data types 12 15
AA-trees 1-5
Algorithm design 13
BinarySearchTree<TermTreeNodeData>
DAGs 45 89 134
BinarySearchTree<TermTreeNodeData>
New line character 22-25 68 207
Nested loops 56 123-126

Figure 1: Binary Search Tree structure of Index.

2 Implementation

You are given BinarySearchTree template class which is almost complete. In order to implement the Index tree and Term trees you will use this class template. The classes IndexNodeData and TermTreeNodeData are defined in the following sections. Once you have those classes you can define Index and Term trees as follows:

```
BinarySearchTree<IndexNodeData> Index;

BinarySearchTree<TermTreeNodeData> tree;
```

Some remarks for the class definitions are given in the following section. You can refer to C++ files for detailed information about class members. In the given files, some methods are left blank for you to implement according to the explanations commented in the method definitions.

2.1 class IndexNodeData

This class describes nodes in Index tree. There are two private data members: a letter (A, B, C, ...) and the term tree.

```
class IndexNodeData{

  private: // data
    char letter;
    BinarySearchTree<TermTreeNodeData> tree;

  public:
    /* public IndexNodeData methods */
  private:
    /* private utility methods */
};
```

IndexNodeData class is completely implemented in IndexNodeData.h and IndexNodeData.cpp files. Therefore, changing the content of these files is not allowed.

2.2 class TermTreeNodeData

This class describes nodes in Term tree. There are two private data members: a term which is a string and a list of page numbers.

```
class TermTreeNodeData {

  private: // data
    string term;
    vector<Pair> pagelist;

  public:
    /* public TermTreeNodeData functions */
```

class Pair is defined to represent page intervals by two integers for start and end page. If start and end page numbers are equal then the interval is accepted as a single page and listed as a single number in the output. Otherwise, they are accepted as page interval and stated as <start>-<end> format in the output as in 1.

TermTreeNodeData and Pair classes are completely implemented in TermTreeNodeData.h and TermTreeNodeData.cpp files. Therefore, changing the content of these files is not allowed.

2.3 class TreeNode

This template class defines the node of a binary tree. It is used by the template class Binary-SearchTree. The class has three private data members: a data variable and two pointers to left and right subtrees.

```
template <class T>
class TreeNode {
  public:
    T data;              // data stored at this node
    TreeNode<T> *left;    // reference to left subtree
    TreeNode<T> *right;   // reference to right subtree

    // Constructs a leaf node with the given data.
    TreeNode<T>(T val=T()) {
      data = val;
      left = nullptr;
      right = nullptr;
    }

    // Constructs a branch node with the given data and links.
    TreeNode<T>(T val, TreeNode<T> *lf, TreeNode<T> *rt) {
      data = val;
      left = lf;
      right = rt;
    }
};
```

2.4 `class BinarySearchTree`

This is a template class to define a binary search tree abstract data type. Its only private data member is `root`, which is a TreeNode pointer to the root element of binary search tree. The class includes template functions to add, remove, search elements in a binary search tree in general. The class also includes copy constructor, destructor and assignment operator. The only empty method in the class is makeItBalanced. You need to to implement this method.

```
template <class T>
class BinarySearchTree {

  public:

    // = operator overloading
    BinarySearchTree& operator=(const BinarySearchTree& t);

    // Adds the given value to this BST in sorted order.
    void add(T value);

    // Removes the given value from this BST while remaining elements
    // stay sorted.
    void remove(T value);

    // Returns if this tree contains value val.
    bool contains(T val);

    // Makes the tree balanced.
    void makeItBalanced();

    /* ... */


  private:
    // Private data member:
    // the overall root of the tree. It is null for an empty tree
    TreeNode<T> *root;

    /* private utility functions */
    void makeItBalanced(TreeNode<T> *&r, vector<T> &A, int n, int &i);

    /* ... */
```

`void makeItBalanced()` restores tree to a minimum height tree, so it will be a balanced tree. It first copies all values in the tree into a vector in inorder traversal. It then deletes all nodes in the tree (you can use the given deleteTree() method here), makes root null and calls the helper method makeItBalanced() (decribed below) to make root point to the root of the new minimum height tree.

`void makeItBalanced(TreeNode *&r, vector<T> &A, int n, int &i)` inserts the values in the given vector into the tree in such a way that the middle value in the vector becomes the root of the new minimum height tree. The algorithm in pseudocode in Algorithm 1.

Note that balancing tree operation is implemented only within these two methods, mostly in `void makeItBalanced(TreeNode *&r, vector<T> &A, int n, int &i)`. In other words, fill the body of these two functions for implementing tree balancing on `BinarySearchTree` class.

You are expected to implement `void makeItBalanced()` and `void makeItBalanced(TreeNode *&r, vector<T> &A, int n, int &i)` functions in `BinarySearchTree.h` file. The other member function definitions are available in the file. You can define additional private utility functions for your solution.

Algorithm 1 makeItBalanced(IntTreeNode *&r, vector <T> &A, int n, int &i)

```
if $n > 0$ then
    // construct the left subtree
    $r \leftarrow$ pointer to new node with NULL child pointers
    makeItBalanced(r->leftChild, n/2)
    // get the root
    // Copy item from the ith position in the vector into r->data
    $i \leftarrow i + 1$
    // construct the right subtree
    makeItBalanced (r->rightChild, (n-1)/2)
end if
```

2.5 class Indexer

`Indexer` is the class that implements the overall index structure. It has one private data member which is the Index tree. It includes methods to perform several operations on the index.

```
class Indexer{

  public:

    void addTerm(string term, Pair p);
    void removeTerm(string term);
    void removeLetter(char letter);
    void makeIndexBalanced();
    void makeLetterBalanced(char letter);


    /* ... */

  private:

    BinarySearchTree<IndexNodeData> Index;
};
```

The class methods are explained in the following:

2.5.1 void addTerm(string term, Pair p)

This function inserts a term together with its page number interval to a Term tree.

The function first checks if the first letter of `term` is in the `Index`. If not, a new node with that first letter is added to `Index`, a new Term tree is created for that letter and the given term is inserted to the new Term tree.
If the letter exists in the Index, this time, `term` is searched in the respective Term tree. If term is not found, then it is inserted as a new node in the Term tree together with the page interval. If

found, the pagelist is updated with the start and end page numbers.

After inserting page number into an existing `TermTreeNodeData`, there is no contiguous or intersecting page intervals in the resulting index. For example, if page numbers 7 and 10 is inserted to a term with 5-8 11-15, the final `pagelist` for the term should be 5-15 as 7-10 interval is intercepting with 5-8 and adjacent to 11-15. Adding new page interval into existing `TermTreeNodeData` element is handled by `TermTreeNodeData& TermTreeNodeData::addPage(Pair p)` function.

2.5.2 `void removeTerm(string term)`

The function removes `TermTreeNodeData` with `term`. If deleted `TermTreeNodeData` is the last term of its corresponding letter, remove that `IndexNodeData` node from `Index`, as well.

2.5.3 `void removeLetter(char letter)`

The function removes IndexNodeData with `letter`.

2.5.4 `void makeIndexBalanced()`

The function makes the overall `Index` a balanced tree. Note that balancing operation is only applied on tree with IndexNodeData elements. In other words, only the nodes with letters is balanced in this function.

2.5.5 `void makeLetterBalanced(char letter)`

The function makes Term tree of a given letter a balanced tree.

You are expected to implement copy constructor, = operator, and these 5 functions above namely, `void addTerm(string term, Pair p)`, `void removeTerm(string term)`, `void removeLetter(char letter)`, `void makeIndexBalanced()`, and `void makeLetterBalanced(char letter)`, in `Indexer.cpp` file. The other member function definitions are available in the file. You can define additional private utility functions for your solution.

3 Regulations

1. **Programming Language:** You will use C++.
2. External libraries other than those already included are not allowed.
3. Those who do the operations (insert, remove, search, print) without utilizing the tree will receive 0 grade.
4. You can change `main.cpp` file to test your implementation.
5. You are allowed to edit `BinarySearchTree.h`, `Indexer.h`, `Indexer.cpp`, and `BinarySearchTree.h` files. Do not modify the content of other homework files.
6. Those who modify already implemented functions and those who insert other data variables or public functions and those who change the prototype of given functions will receive 0 grade.
7. Options used for `g++` are `-ansi -Wall -pedantic-errors -std=c++0x -O0`. They are already included in the provided Makefile.
8. You can add private member functions whenever it is explicitly allowed.

9. **Late Submission:** You have 5 days for late submissions in all assignments. Your assignment will not be accepted if you submit more than 5 days in total.

10. **Cheating:** We have zero tolerance policy for cheating. In case of cheating, all parts involved (source(s) and receiver(s)) get zero. People involved in cheating will be punished according to the university regulations. Remember that students of this course are bounded to code of honor and its violation is subject to severe punishment.

11. **Newsgroup:** You must follow odtuclass (odtuclass.metu.edu.tr) for discussions and possible updates on a daily basis.

4 Submission

- Submission will be done via Moodle (cengclass.ceng.metu.edu.tr).
- Do not write a *main* function in any of your source files.
- A test environment will be ready in Moodle.
  - You can submit your source files to Moodle and test your work with a subset of evaluation inputs and outputs.
  - Additional test cases will be used for evaluation of your final grade. So, your actual grades may be different than the ones you get in Moodle.
  - Only the last submission before the deadline will be graded.