Middle East Technical University | Department of Computer Engineering

CENG 213

Data Structures

Fall '2016-2017

Programming Assignment 2

Due date: 24 December 2016, Saturday, 23:55

1 Objectives

This assignment aims to make you familiar with binary search tree implementation to build search indices for generic item collections. You will devise a proper method to reorder the balance back to the tree structure based on a rather flexible condition.

Keywords: *Binary Search Tree, Balanced Complete Tree*

2 Tree Class Template (50pts)

You will implement a binary search tree template for indexing generic items under different comparision schemes. Tree, a set of connected nodes via pointer links such that no simple cycle should exist overall, will attain a single designated node at a special position called root, from which left and right subtrees are to be recursively constructed. Each node will hold a pointer variable addressing a particular item in the collection together with pointers to roots of nodes in its left and right subtrees as well as a link to its parent node, respectively. In this task, you will utilize C++ STL `list` as the ADT implementation of your item collection and upon this you will construct indexed trees whose functionality will be described in sufficient detail subsequently. You can use STL `queue` and STL `stack` for required computations. However, other than these no STL use is allowed.

2.1 Private Data

The class has a designated pointer to the root of the tree and a function object to carry out comparison to yield whether an item is less than another, together with variables to store the number of nodes in the tree and to indicate whether the tree is currently balanced or not.

```
TreeNode * root; //designated root

/* isLessThan( const Comparable & lhs, const Comparable & rhs )
 * returns true if lhs<rhs and false otherwise, keep reading for details */
Comparator isLessThan;

size_t size; //number of nodes
bool balanced;
```

Note that a node of tree is implemented as a `struct` construct inside `private` section of the class.

```
struct TreeNode
{
    const Comparable * element; //adress of the item in the list , data in the list cannot be modified
    TreeNode * left; //root node of left subtree
    TreeNode * right; //root node of right subtree
    TreeNode * parent; //address of the parent node
    size_t height; //distance to deepest leaf

    /* constructor , all data parts can be directly set as well */
    TreeNode( const Comparable * el,
              TreeNode * l, TreeNode * r,
              TreeNode * p )
      : element( el ),
        left ( l ), right( r ),
        parent( p )
    {
    }
};
```

Copy constructor and assignment operator signatures are also included in the `private` section. We will not allow their use in this assignment and since compiler injects a default version of the two performing shallow copying in their presence, we have provided their prototype and you should perform no implementation. This is a trick utilized in C++ world so as to make objects of the class type uncopiable.
You can add other variables and functions to the private part of the class. For a neat application of recursion, you should define utility functions in this section. Follow the design example of overloaded `print` method in both `public` and `private` sections. Maintaining the programming approach for Binary Search Tree implementation in the textbook is highly recommended.

2.2 Interface

Public interface of the class shows its capabilities. Do not modify the signatures of methods in `public` section. In this assignment implementations of `findMin` , `findMax`, `contains`, `isEmpty`, `getSize`, `getHeight`, `isBalanced`, and `print` methods have already been completed. You are expected to implement the following `public` section functions.

2.2.1 Tree( );

The default constructor is used to initialize an empty tree of size zero, which is balanced and has `NULL` as its pointer to the root.

2.2.2 Tree(const list <Comparable> &);

Assuming that the item list is sorted in ascending order with respect to `isLessThan` function object provided as a class template argument whose default is the ordinary `operator<`, this one-argument constructor builds up a complete tree comprising of number of nodes equal to the size of the list and populates its pointer element values in accordance with the inorder traversal of the tree and the order of items in the list. In other words, here you are expected to create a minimum-height binary search tree that is complete with its nodes holding pointers to corresponding sorted items in the list. Determine the height of each node as well.

2.2.3 ~Tree( );

Destructor is responsible for releasing all dynamically allocated memory associated with node pointers.

2.2.4 void construct(const list <Comparable> &);

This method will restore the tree into an empty tree state and then will build a minimum-height complete binary search tree based on the values of items stored in the list, just exactly the same way as the one-argument constructor operates. Determine the height of each node as well.

2.2.5 const Comparable * getAddress(const Comparable &) const;

This method returns the pointer variable holding the address of the list item with the specified values. If the variable does not exist, then a `NULL` pointer should be returned.

2.2.6 void insert (const Comparable *);

Using this method, one can grow the size of the indexed tree in question. Correct position to insert the new pointer-to-item should be revealed by the binary search tree property. Note that following insertion heights of the nodes residing on the path from the new node to the root may be altered. Update the corresponding height attributes of these nodes.
It is well-known that insertion into the traditional binary search tree might create height imbalance between certain subtrees, causing the search time to increase into linear complexity. To prevent such cases, you must check whether the newly inserted node results in **height imbalance** implied by,

$$height(T) \geq 2(\lfloor \log_2 N \rfloor) \ , N > 1 \tag{1}$$

where T stands for the tree structure, and N is the size of the tree, i.e. the number of nodes in the tree. The two other cases not covered by the predicate in (1) is that a tree with one node is balanced in which $N = 1$ and that empty tree is also balanced, i.e. for $N = 0$. Consequently other than these two cases, namely whenever the predicate in (1) holds true, we claim that the tree is considerably out-of-balance and rebalancing is required.
So as to rebalance the tree, you must get pointers to items in sorted order via inorder traversal into some auxiliary list, and rebuild the tree into a minimum-height complete binary search tree with size being equal to the number of items in the list, just as done in the one-argument constructor. Also you must populate the nodes with corresponding item pointers while obeying the binary search tree property.
This is a linear operation, which we perform only when tree is regarded as out of balance. In other words, as opposed to AVL search tree we allow for higher imbalance factors bounded by another balance condition so that average search cost for our tree structure will not increase into linear complexity. Moreover, for the new tree we have a more flexible balance condition whose parameters can be varied without directly changing the followed procedure to restore balance. Yet, for AVL trees increasing allowed imbalance even by one would result in more rotations until the tree is balanced again.

2.2.7 void remove(const Comparable *);

This method follows the removal algorithm of the binary search tree ADT. Inspect and utilize the implementation of removal routine of the book with essential modifications specific to this design. As in case of insertion, remove operation may alter the height information of nodes on the path from the deleted node up to the root. Update height attributes of all such nodes.
Removal can also alter the balance of the tree. For this operation as well, you must check whether an

imbalance in height is incurred by checking the truth of the predicate in (1) and reorganize current tree into the minimum-height complete binary search tree by following the rebalancing operation specified under insert method.

3 MusicList (50pts)

In this assignment your task will be implementing a collection of songs with tree indices built on top of distinct song attributes. For this reason, the aggregate class type called `Song` has been provided, in which a song is represented by five attributes: the name of the song, the band/artist that composed the song, the year in which the song is published, the length of the song in seconds, and a boolean attribute to state whether the song is currently active or not, which will be used by the deletion procedure.
Constructor, destructor and accessor methods together with a mutator function to change the activeness status of the song have been provided. Inside constructor the first three parameters to initialize the first three attributes must be present, and when omitted as the fourth parameters, length of the song in seconds will be defaulted to zero. Each song has its activeness attribute set to true upon construction, which can be changed into a false later on, in which it permanently preserves this value until destruction.
Lastly, `operator<<` has been overloaded for `Song` class objects, by which values of attributes can be output on `ostream` type objects such as the `cout` in `std` namespace of `iostream` library. Observe the implementation of `Song` class, but do not modify any piece of code included in corresponding files.

3.1 NameComparator and YearComparator

Templated tree described in previous section requires a template object of a certain class type that provides the functionality of comparing two items in order to initialize pointers stored in tree nodes correctly. You can overload `operator<` in only one way. However, for our task and for most tasks, more than one way to carry out the comparison to deem whether an object is less than another is required since we would like to compare songs based on different attributes. C++ utilizes *function objects* to achieve that, and also most STL functionality such as `std::list::sort` accepts function objects to sort entries with respect to various criteria. For more information on function objects, you can refer to section 1.6.4 of your textbook.
As a very straightforward application of function objects within our problem, a new class type must be defined with the overloaded paranthesis operator taking two objects of `Song` type by constant reference, and returns a boolean value of true if the first one is smaller than the second, or false otherwise. Through this technique we can devise arbitrary ways to objects of the same type.
For the purposes of this task, you must define `NameComparator` type to compare two songs based on their names and state whether the first one comes before the second with respect to only names. The outline of the class is included subsequently.

```
class NameComparator  //compare songs based on their names
{
public:    //only one public function: overloaded paranthesis
  bool operator( )( const Song & s1, const Song & s2 ) const
  {
    /* Complete this part */
  }
};
```

Your implementation of `NameComparator` must be based on the following rules.

- If the name of the first song comes lexicographically before the second one, return true.
- If the second song comes alphabetically after the first one, then return false.
- If the two songs have the same name, compare the band names to determine the return type.
- If there is still no difference with respect to these attributes, compare the years to determine the return type.
- For other cases, return false.

You must define another class called `YearComparator` so that you may compare two songs based on the years in which they were published. The outline is included in the following.

```
class YearComparator  //compare songs based on years
{
public: //only one public function: overloaded paranthesis
  bool operator( )( const Song & s1, const Song & s2 ) const
  {
    /* Complete this part */
  }
};
```

Your implementation of `YearComparator` must carry out its computation according to following rules.

- If the first song was published before the second one, return true.
- If the second song is an older song than the first one, return false.
- If two songs are published in the same year, compare the band names lexicographically to return an appropriate boolean value.
- If the two songs are composed by the same band in the same year, compare song names in lexicographic order to determine the return type.
- Otherwise, return false.

Through the use of `NameComparator` and `YearComparator` as the second template parameter in the data structure Tree, you will create name-based and year-based indices for the collection of songs.

3.2 MusicList Private Data

Your list of songs are encapsulated into the `MusicList` class, which internally stores song collection via the doubly-linked `list` in C++ STL. Since searching through linked list is a linear time operation, you are going to declare and use two Tree objects, holding the addresses of nodes in the linked list with respect to `NameComparator` and `YearComparator` function objects.

```
list <Song> playlist; //STL list is adopted as it can dynamically grow or shrink without validating ↩
    existing nodes' addresses
Tree<Song,NameComparator> nameBasedIndex; //search for names efficiently
Tree<Song,YearComparator> yearBasedIndex; //search for years efficiently

NameComparator nameLessThan; //used for conducting name-based comparison of songs to sort the ↩
    list initially
YearComparator yearLessThan; //used for conducting year-based comparison of songs to sort the list ↩
    initially
```

All retrieve operations will be queried on corresponding trees. Assume that **no more than one** song exists with the same name made by the same band in the same year. In other words, duration will not be used to differentiate among songs. However, two songs may share the same information on any two of the name, band and year attributes.
You can define additional private data members and methods depending on the needs of your design.

3.3 MusicList Interface

You must implement the following `public` interface functions. Do not change the prototypes of these functions.

3.3.1 MusicList( );

Constructs an empty list of songs and two empty trees for name-based and year-based indices.

3.3.2 MusicList(const list <Song>&);

Initializes the playlist based on the list argument. If this list is not sorted in name attribute with respect to `NameComparator` object then sort the list and construct the name-based index based on the sorted list. Similarly, ensure that the list is sorted with respect to `YearComparator` object and construct-the year-based index based on this list.

3.3.3 ~MusicList( );

Erase all songs marked as not active on the list before destroying the list and the trees.

3.3.4 void insert (const string &, const string &, int , size_t =0);

Based on the parameters of the function, define a new `Song` variable. If the song already exists in the indices, return immediately. Otherwise, append the song at the end of the playlist and retrieve a pointer to its location within the list to be inserted into both the name-based index and the year-based index.

3.3.5 void remove(const string &, const string &, int);

Based on the given attributes, if such a song exists in tree indices, set the data of corresponding node in the linked list as not active, and remove the pointers from both name-based and year-based tree indices. If two times the number of unactive nodes in the linked list is equal to or greater than the size of the playlist, perform a linear traversal of the linked list to physically erase all unactive nodes.

3.3.6 int getNumberOfSongs( ) const;

This method must return the number of songs currently included in the playlist, excluding the deleted ones.

3.3.7 void printAllNameSorted( ) const;

Via name-based tree index perform a suitable tree traversal so that you will print all active songs in the playlist sorted in accordance with the `NameComparator` object on the `cout` object. After each song, print an `endl` I/O manipulator.

3.3.8 void printAllYearSorted( ) const;

Via year-based tree index perform a tree traversal so as to print all active nodes in the playlist sorted according to the regulations of `YearComparator` object onto the `cout` object. Following each song, print an `endl` I/O manipulator.

3.3.9 void printNameRange(const string &, const string &) const;

The two parameters of this method specify lower and upper limits for the name attribute so that you will print all songs whose names fall into the specified range. Here you should not be visiting all tree nodes over and over again. Try to devise a method so that only the nodes sent to the output stream will be visited in one pass. Print each song in a single line on `cout`.

3.3.10 void printYearRange(int , int) const;

Here the two parameters represent lower and upper bounds to construct a closed range of years so that all songs published in this interval will be print out to the console each in a dedicated line. Regarding tree traversal obey the same restrictions of `printNameRange` method.

4 Driver programs

For testing `Tree` functionality, a driver program under the name `main_tree_ex*.cpp` will be provided with together with the correct outputs.
Similarly for testing of `MusicList` functionality, a driver program with name `main_musiclist_ex*.cpp` will be provided with sample inputs and correct outputs will be distributed. In the days to come expect more test cases, also you can share your own inputs and outputs via the COW newsgroup.

5 Regulations

1. **Programming Language:** You will use C++.
2. **S**tandard Template Library is allowed only for `list`, `stack`, and `queue`.
3. **E**xternal libraries are not allowed.
4. **T**hose who do search, update, remove operations directly on the STL `list` with no use of tree indices will receive 0 grade.
5. **Late Submission:** You have 3 days for late submission.
6. **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.
7. **R**emember that students of this course are bounded to code of honor and its violation is subject to severe punishment.
8. **Newsgroup:** You must follow the newsgroup (news.ceng.metu.edu.tr) for discussions and possible updates on a daily basis.

6 Submission

- Submission will be done via Moodle.
- 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, and only the last submission before the deadline will be graded.