Objective: To implement a Binary Tree, a simple map class using hash table and C++ class template.

Specification and Requirement:
1. You need to read in two files. One is variable value table that stores names of variables and its values separated by space as shown below. The other is a list of expressions that each expression is separated by line break and each term within each expression is separated by space. You need to read in variables and make a hash table to store values. You need to build a simple hash map with your own hash function and collision resolution strategy. (30% of total grade)

<Variables.txt>
abc 10
bb 12
ac 5
ef 7
ag1 5
ag2 30
// You may assume that the file name is “Variables.txt” and all the values are positive integers.

<Expressions.txt>
( abc + bb ) * ac – ef / ag1
abc * bb * bb – abc *ag2
( ag1 – ag2 ) / ac
ef + ac / bb + ac * ac
ac = bb
ac / ag2
// You may assume each term should exist at variables table.
// ac = bb should change the value of the ‘ac’, so the expression ac/ag2 should use the updated value. Also ac = bb also generates a value of ac that needs to be inserted to the heap.

(HINT)
You may use a string tokenizer to tokenize each token.

```cpp
#include <string>
#include <algorithm>
#include <vector>

void Tokenize(const string& str, 
              vector<string>& tokens, 
              const string& delimiters = " ")
{
    // Skip delimiters at beginning.
    string::size_type lastPos = str.find_first_not_of(delimiters, 0);
    // Find first "non-delimiter".
    string::size_type pos = str.find_first_of(delimiters, lastPos);

    while (string::npos != pos || string::npos != lastPos) 
    {
```
// Found a token, add it to the vector.
tokens.push_back(str.substr(lastPos, pos - lastPos));
// Skip delimiters. Note the "not_of"
lastPos = str.find_first_not_of(delimiters, pos);
// Find next "non-delimiter"
pos = str.find_first_of(delimiters, lastPos);
}
}

Usage sample
string str(rec);
vector<string> tokens;
Tokenize(str, tokens);
copy(tokens.begin(), tokens.end(), ostream_iterator<string>(cout, ", "));
vector<string>::iterator i = tokens.begin();

2. You need to build a binary tree for each expression and traverse the tree and print the expression
in both prefix order and postfix order. After printing, you need to evaluate the binary tree to
generate the computation result using the hash map of the variables. Build your own binary tree
using pointer-based approach (not array based) (30% of total grade)

3. The results need to be inserted into a (max) heap. You should not use STL make_heap for this and
you need to make your own heap class using array. After constructing the heap, print the heap.
Remove one root and print the heap after the removing the root. (30% of total grade)

4. Documentation is required to explain your classes. (10% of total grade)

5. Hints for hash map. You may use your own way to implement hashmap. This is only a
recommendation.

template<class KeyType, class ValType>
class Item
{
    public:

        // default constructor
    Item(): key(KeyType()), value(ValType())
    {}

        // constructor
    Item(const KeyType& v1, const ValType& v2): key(v1), value(v2)
    {}

        // data members : 2 template data types
    KeyType key;
    ValType value;
};
template <class Key, class Val, class Func>
class SimpleMap
{
    public:

    // Constructor: create an empty hash table with SimpleMapSize
    // Call vector's constructor to initialize hash table
    SimpleMap(int SimpleMapSize = 1024);

    bool empty() const;  // check if numItems = 0 ?
    int size() const;    // return numItems

    // Goal: insert new item = (key, value) into hash table
    // if key is already in the table, return false
    // else, insert new item and return true
    bool insert(const Key& key, const Val& value);

    // Goal: delete item with the given key from hash table
    // if key is not in the table, return false
    // else erase item and return true
    bool erase(const Key& key);
// Goal: find item with the given key from hash table
// if the item is in the table,
// set value = the item's value and return true
// else return false
bool find (const Key& key, Val& value) const;

// Goal: This is an important operation in map. It supports
// simple interface, i.e. using operator [], to insert and
// retrieve item. Refer to test program for examples
//
// if key is in the table, return the reference to the value
// else, insert an item = (key, Val()) and return the reference
// to the value. Note: Val() = default constructor
Val& operator[] (const Key& key);

// This function displays all items in hash table
// Use separator to separate items from a same bucket
void display(const string separator = "|") const;

private:

    int  hashTableSize;  // number of entries in hash table
    Func hashFunc;       // hash function object
    int  numItems;       // current number of item in hash table

    // a hash table is a dynamic array. Each entry is a list of items
    // an item contains a constant key and a value, i.e. key cannot be
    // change

    vector<list<Item<const Key, Val> > > hashTable;
};