Homework 4: Implementing Ordered Associative Array

DRAFT: Open for comment in the discussion forum.

Educational Objectives: On successful completion of this assignment, the student should be able to

- Define the concept of associative container as a re-usable component in programs
- State the distinction between unimodal and multimodal associative containers, and:
  ☐☐☐ Give examples of each type
  ☐☐☐☐ Describe use cases making each type appropriate
- State the distinction between ordered and unordered associative containers
  ☐☐☐ Give examples of each type
  ☐☐☐☐ Describe use cases making each type appropriate
- State the API for associative containers of these types:
  ☐☐☐ Unimodal Ordered Map (aka Ordered Table)
  ☐☐☐☐ Multimodal Ordered Map (aka Ordered Multimap)
  ☐☐☐☐☐ Unimodal Unordered Map (aka Unordered Table)
  ☐☐☐☐ Multimodal Unordered Map (aka Unordered Multimap)
  ☐☐☐ Ordered Associative Array
  ☐☐☐☐ Unordered Associative Array
  Describe the behavior and state the runtime expectations for each operation.
- Describe various implementation plans for ordered associative containers, and discuss whether and why runtime expectations are met by the implementation.
- Implement Ordered Associative Array as a left-leaning red-black binary search tree.
- Use the Ordered Associative Array API to refactor applications that were originally designed around the Ordered Set of Pairs API.

Background Knowledge Required: Be sure that you have mastered the material in these chapters before beginning the assignment:
Introduction to Sets, Introduction to Maps, Binary Search Trees, and Balanced BSTs.

Operational Objectives: Create an implementation of the Ordered Associative Array API using left-leaning red-black trees. Illustrate the use of the API by refactoring your `WordBench` as a client of Ordered Associative Array API.

Deliverables:

```
oaa.h           # the ordered associative array class template
wordbench2.h    # defines wordbench refactored to use the OAA API
wordbench2.cpp  # implements wordbench2.h
log.txt         # your standard work log
```

Procedural Requirements

☐☐☐ Keep a text file log of your development and testing activities in `log.txt`.

☐☐☐ Begin by copying all of the files from the assignment distribution directory, which will include:

```
hw4/main2.cpp       # driver program for wordbench2
hw4/foaa.cpp        # functionality test for OAA
hw4/rantable.cpp    # random table file generator
hw4/makefile        # makefile for project - builds wb2.x and foaa.x
hw4/hw4submit.sh    # submit script
```

☐☐☐ Define and implement the class template `OAA<K,D>`, placing the code in the file `oaa.h`.

☐☐☐ Thoroughly test your `OAA<>` with the distributed test client programs `foaa.cpp` and `moaa.cpp`. Be sure to log all test activity.

☐☐☐ Define the application WordBench, refactored as a client of `OAA<fsu::String, size_t>`, in the header file `wordbench2.h`, and implement the refactored WordBench in the file `wordbench2.cpp`

☐☐☐ Test your refactored WordBench thouroughly to be certain that it is a true refactoring or the original. (Refactoring is defined to be re-coding without changing the program behavior.) Again, log all test activity.

☐☐☐ Be sure to fully cite all references used for code and ideas, including URLs for web-based resources. These citations should be in the file documentation and if appropriate detailed in relevant code locations. Also cite all resources used in your log.

☐☐☐ Be sure to fully cite all references used for code and ideas, including URLs for web-based resources. These citations should be in two places: (1) the code file documentation and if appropriate detailed in relevant code locations; and (2) in your log.

☐☐☐ Submit the assignment using the script `hw4submit.sh`.

Warning: *Submit scripts do not work on the program and linprog servers. Use shell.cs.fsu.edu to submit assignments. If you do not receive the second confirmation with the contents of your assignment, there has been a malfunction.*

Requirements - Ordered Associative Array

☐☐☐ The following definition should be used:

```
template < typename K , typename D , class P = LessThan<K> >
class OAA
{
public:

  typedef K    KeyType;
  typedef D    DataType;
  typedef P    PredicateType;

           OAA   ();
  explicit OAA   (P p);
           ~OAA  ();

  DataType& operator [] (const KeyType& k) { return Get(k); }

  void    Put    (const KeyType& k , const DataType& d);
  D&      Get    (const KeyType& k);
  void    Clear  ();

  bool    Empty  () const;
  size_t  Size   () const;
  int     Height () const;

  template <class F>
  void    Traverse (F f) const { RTraverse(root_,f); }

  void    Display  (std::ostream& os, int cw1, int cw2) const;

  void    Dump     (std::ostream& os) const;
  void    Dump     (std::ostream& os, int cw) const;
  void    Dump     (std::ostream& os, int cw, char fill) const;

  enum Flags { ZERO = 0x00 , DEAD = 0x01, RED = 0x02 , DEFAULT = RED };
  static const char* ColorMap (unsigned char flags)
  {
    switch(flags)
    {
      case 0x00: case 0x01: return ANSI_COLOR_BOLD_BLUE;  // bits 00, 01
      case 0x02: case 0x03: return ANSI_COLOR_BOLD_RED;   // bits 10, 11
      default: return "unknown color";   // unknown flags
    }
  }

private: // definitions and relationships

  class Node // vertex in the tree structure
  {
    const KeyType   key_;
          DataType  data_;
    Node            * lchild_,
                    * rchild_;
    unsigned char   flags_;
    Node (const KeyType& k, const DataType& d, Flags flags = DEFAULT)
      : key_(k), data_(d), lchild_(0), rchild_(0), flags_(flags)
    {}
    friend class OAA<K,D,P>;
    bool IsRed    () const { return 0 != (RED & flags_); }
```

```
        bool IsBlack  () const { return !IsRed(); }
        void SetRed   ()       { flags_ |= RED; }
        void SetBlack ()       { flags_ &= ~RED; }
      }; // internal class Node

      class PrintNode // function class facilitates Display()
      {
      public:
        PrintNode (std::ostream& os, int cw1, int cw2) : os_(os), cw1_(cw1),
cw2_(cw2)
        {}
        void operator() (const Node * n) const
        {
          os_ << std::setw(cw1_) << n->key_ << std::setw(cw2_) << n->data_ << '\n';
        }
      private:
        std::ostream& os_;
        int cw1_, cw2_;
      }; // internal function class PrintNode

    private: // data
      Node *         root_;
      PredicateType  pred_;

    private: // methods
      static Node * NewNode(const K& k, const D& d, Flags flags = DEFAULT);
      static void   RRelease(Node* n); // deletes all descendants of n
      static size_t RSize(Node * n);
      static int    RHeight(Node * n);

      template < class F >
      static void RTraverse (Node * n, F f);

      // recursive left-leaning get
      Node * RGet(Node* nptr, const K& kval, Node*& location);

      // rotations
      static Node * RotateLeft(Node * n);
      static Node * RotateRight(Node * n);

    private: // copy facilitation - do not implement
      OAA   (const OAA& a);  // copy constructor
      OAA& operator= (const OAA& a); // assignment operator
    }; // class OAA<K,D>
```

Note that the implementations of all OAA methods are discussed in the lecture notes in one form or another.

☐☐☐ It is worth pointing out what is NOT in these requirements that would be in a "full" OAA API:

- ☐☐☐ Object comparison operators == and !=
- ☐☐☐☐ Copy constructor and assignment operator
- ☐☐☐☐☐ Iterators and iterator support
- ☐☐☐☐ Remove or Erase

The remaining portion of the OAA API consist of Get, Put, Clear, constructors and destructor -- arguably the minimal necessary for a useful container.

☐☐☐ Note that the AA bracket operator is in the interface and is implemented in-line above with a single call to Get. Also note that Put can be implemented with a single call to Get, which leaves Get as the principal functionality requiring implementation.

☐☐☐ The various const methods measure useful characteristics of the underlying BST and provide output useful in the development process as well as offering client programs insight into the AA structure.

☐☐☐ The color system is outlined here just as in the lecture notes. The ColorMap is used by the Dump methods to color nodes at output. Color is manipulated by the four Node methods for detecting and changing node color.

☐☐☐ The various "private" statements are redundant, but they emphasize the various reasons for using that designation: (1) to have private in-class definitions, such as Node or typedef statements, and to record any friend relationships that might be needed; (2) private data in the form of variables; (3) private methods; and (4) things that are privatized to prevent their use.

Requirements - WordBench

☐☐☐ Here is a working header file for the refactored WordBench:

```
/*
    wordbench2.h
*/

#include <xstring.h>
#include <list.h>
#include <oaa.h>

class WordBench
{
public:
  WordBench           ();
  virtual ~WordBench  ();
  bool   ReadText     (const fsu::String& infile);
  bool   WriteReport  (const fsu::String& outfile, unsigned short c1 = 15,
unsigned short c2 = 15) const;
  void   ShowSummary  () const;
  void   Erase        ();

private:
  typedef fsu::String              KeyType;
  typedef size_t                   DataType;

  size_t                           count_;
  fsu::OAA  < KeyType , DataType > frequency_;
  fsu::List < fsu::String >        infiles_;
  static void Cleanup  (fsu::String& s);
} ;
```

The set "wordset_" from the original design is replaced with the ordered associative array "frequency_".

☐☐☐ Note the private terminology is changed slightly. (Of course, the API is not changed.) The main storage OAA is called `frequency_` which makes very readable code of this form:

```
...
Cleanup(str);
if (str.Length() != 0)
{
  ++frequency_[str];
  ++numwords;
} // end if
...
```

This snippet is the inner core of the processing loop implementing ReadText. The main loop implementing ReadText is now only 5 lines of code.

☐☐☐ Another small change is that it is no longer possible to loop through the data to count the words, because we are not defining an Iterator class. We could work out a way to make this count using a traversal with a special function object that retrieves the specific frequencies, but it is simpler just to have a class variable `count_` that maintains the total number of words read (and is reset to 0 by Clear()).