CS242: Object-Oriented Design and Programming

Program Assignment 4
Part 1 (Ordered List) Due Saturday, March 1st, 1997

An Ordered List is an *Abstract Data Type* (ADT) that implements a priority queue with an ordering relation defined between each pair of elements in the list. Common operations on an Ordered List include *insert*, *remove*, *find*, *front*, *is_empty* and *is_full*. This part of your programming assignment focuses upon building and using *unbounded* implementations of Ordered Lists.

You will implement the following version of the ADT Ordered List:

1. *Unbounded Ordered List* – this implementation will use a circular linked list of items of type `T`, which is "unbounded" and uses dynamic memory. Each type `T` will contain comparison methods (*i.e.*, `operator <` and `operator==`) that will be used to keep the list ordered by "ascending" values. Duplicates item values are allowed in this implementation.

Here's the interface that you'll use for the unbounded Ordered List:

```
#if !defined (_ORDERED_LIST)
#define _ORDERED_LIST

/* -*- C++ -*- */
#include <stdlib.h>

#if defined (__GNUC__)
#define THROW(X)
#else
#define THROW(X) throw X
#endif /* __GNUC__ */

#include "Exceptions.h"

template <class T>
class Ordered_List_Iterator
  // = TITLE
  //     Base class for iterating over an Ordered_List.
{
public:
  virtual int get (T &item) = 0;
  // Gets the next unseen item in the Ordered_List and advances the iterator.
  // Returns -1 when all <cur_size> items have been seen, else 0.

  virtual int set (const T &item) = 0;
  // Sets the next item in the Ordered_List and advances the iterator.
  // Returns -1 when all <cur_size> items have been seen, else 0.

private:
  // You fill in here as needed.
};

// Forward declaration.
template <class T>
class Ordered_List_Node
  // = TITLE
  //     Abstract base class that defines a generic
  //     FIFO abstract data type.
  //
  // = DESCRIPTION
  //     This implementation of a Queue uses a
  //     circular linked list.
{
  friend class Ordered_List<T>;
  friend class Ordered_List_Iterator<T>;
protected:
  // = Initialization methods
  Ordered_List_Node (const T &item,
                     Ordered_List_Node<T> *next = 0);

  void *operator new (size_t bytes);
  // Allocate a new <Ordered_List_Node>, trying first from the
  // <free_list_> and if that’s empty try from the global <::operator
  // new>.

  void operator delete (void *ptr);
  // Return <ptr> to the <free_list_>.

  static void free_list_allocate (size_t n);
  // Preallocate <n> <Ordered_List_Nodes> and store them on the
  // <free_list_>.

  static void free_list_release (void);
  // Returns all dynamic memory on the free list to the free store.

private:
  static Ordered_List_Node<T> *free_list_;
  // Head of the "free list", which is a stack of
  // <Ordered_List_Nodes> used to speed up allocation.

  static Ordered_List_Node<T> *last_resort_;
  // This is the node of last resort if memory allocation fails...

  T item_;
  // Item in this node.  Must have operator < and operator ==
  // defined on it.

  Ordered_List_Node<T> *next_;
  // Pointer to the next node in the list.
};

template <class T>
class Ordered_List
  // = TITLE
  //     Base class that defines a generic
  //     Ordered List abstract data type.
{
public:
  typedef T TYPE;
  // C++ trait.

  // Overflow and underflow exception.
  class Underflow {};

  // = Construction and termination methods.

  Ordered_List (size_t size_hint);
  // Constructor.

  virtual ˜Ordered_List (void);
  // Perform actions needed when Ordered_List goes out of scope.

  // = Classic Ordered_List operations.
  virtual void insert (const T &new_item)
    throw (Memory_Error);
  // Place a <new_item> into the list so that it is < or ==
  // to all nodes that are greater than it.

  virtual int find (const T &item);
  // Returns 1 if <item> is in the list, else 0.

  virtual int remove (const T &item);
  // Remove the <item> in the <Ordered_List>.  Returns 0 on
  // success and -1 on failure.

  virtual T front (void) throw (Underflow);
  // Returns the front Ordered_List item without removing it.

  // = Check boundary conditions for Ordered_List operations.

  virtual int is_empty (void) const;
  // Returns 1 if the Ordered_List is empty, otherwise returns 0.

  virtual int is_full (void) const;
  // Returns 1 if the Ordered_List is full, otherwise returns 0.

  // = Factory method for creating the right iterator.

  virtual Ordered_List_Iterator<T> *iterator (void) const throw (Memory_Error);
  // Returns a newly allocated iterator (this must be deleted by the
  // caller).  Throws <Memory_Error> if allocation fails.

  virtual size_t size (void) const;
  // Returns the current number of elements in the Ordered_List.

  // = Template methods.

  virtual int operator == (const Ordered_List<T> &s) const;
  // Checks for Ordered_List equality.

  int operator != (const Ordered_List<T> &s) const;
  // Checks for Ordered_List inequality.

  virtual void operator= (const Ordered_List<T> &rhs);
  // Copy constructor.

  static void free_list_release (void);
  // Returns all dynamic memory on the free list to the free store.

private:
  Ordered_List_Node<T> *tail_;
  // We only need to keep a single pointer for the circular
  // linked list.  This points to the tail of the Ordered List.
  // Since the list is circular, the head of the list is always
  // this->tail_->next_;

  size_t count_;
  // Number of items that are currently in the list.
};

#endif /* ORDERED_LIST */
```

Test Driver Code

The following code implements a test driver to test your `Ordered_List` implementation:

```
// Use an Ordered_List to print a name.
#include <iostream.h>
#include <assert.h>
#include "Ordered_List.h"

#define ORDERED_LIST Ordered_List<char>

int
main (int argc, char *argv[])
{
  const int MAX_NAME_LEN = 80;
  ORDERED_LIST::TYPE name[MAX_NAME_LEN];
  ORDERED_LIST *o1 = new ORDERED_LIST (MAX_NAME_LEN);

  cout << "Please enter your name..: ";
  cin.getline (name, MAX_NAME_LEN);
  size_t readin = cin.gcount () - 1;

  for (size_t i = 0; i < readin && !o1->is_full (); i++)
    o1->insert (name[i]);

  assert (o1->size () == readin);

  // Test the copy constructor.
  ORDERED_LIST o2 (*o1);
  ORDERED_LIST o3 (o2);
  assert (*o1 == o2);
  assert (o2 == o3);

  // Test the assignment operator
  *o1 = o2;
  assert (*o1 == o2);

  // Test the assignment operator
  *o1 = o3;
  assert (*o1 == o3);

  cout << "your name is..: ";

  try
    {
      for (ORDERED_LIST::TYPE last = '\0';
           !o1->is_empty ();
          )
        {
          ORDERED_LIST::TYPE c = o1->front ();
          cout << c;
          assert (last < c || last == c);
          assert (o1->find (c));
          assert (o1->remove (c) != -1);

          // Check for duplicates.
          if (o1->find (c))
            cout << "duplicate of " << c << " exists" << endl;
        }
    }
  catch (Memory_Error)
    {
      cerr << "new failed" << endl;
    }

  cout << endl;
  assert (o1->is_empty ());
  assert (!o2.is_empty ());
  assert (!o3.is_empty ());
  assert (*o1 != o2);
  assert (*o1 != o3);
  delete o1;

  // Release all the dynamic memory.
  ORDERED_LIST::free_list_release ();
  return 0;
}
```

Getting Started

You can get the “shells” and Makefile for part one of the program from your account on cec. These files are stored in `/project/adaptive/cs242/assignment-4/`. Here’s a script that shows you how to set everything up and get these files:

```
% cd ~/cs242
% mkdir assignment-4
% cd assignment-4
% cp -r /project/adaptive/cs242/assignment-4/* .
% ls
Makefile
list-test.C
Ordered_List.C
Ordered_List.h
% make
```

The `Makefile`, `list-test.C` and various header files are written for you. All you need to do is edit the `*.C` files to add the methods that implement the bounded and unbounded `Ordered_Lists`.