Templates Assignment

Many of you will have some experience using the collections of the C++ standard library. The design of such libraries is a complex undertaking, with many tradeoffs to be considered—far more than can be covered in this course. The current assignment will give you a taste for some of the issues, and will help to familiarize you with generics (templates) and with the synergy between subtype and parametric polymorphism.

The specifics

Your task is to implement variants on a `set` collection type. These should look familiar given our discussion of the dictionary (mapping) types used by `switch` (`case`) statements.

I am providing a file of (incomplete) starter code. Its most basic definition is of the abstract class `simple_set`:

```
template<class T, class C = comp<T>>
class simple_set {
  public:
    virtual ~simple_set<T, C>() { }
    virtual simple_set<T, C>& operator+=(const T item) = 0;
        // add item to set
    virtual simple_set<T, C>& operator-=(const T item) = 0;
        // remove item from set, if it was present
    virtual bool contains(const T& item) const = 0;
        // indicate whether item is in set
};
```

Class `C` is a "comparator"; it provides an `equals` method that can be used to identify duplicate elements. (It also provides a `precedes` method, but that isn't needed for `simple_set`.) The default comparator `comp` employs `T`'s `operator==`, assuming it exists. You might want a different comparator for, say `char*`s, since the built-in == compares pointer values, not lexicographic ordering of strings.

As an example, the starter code contains a concrete class `std_simple_set` that adapts the sets of the standard library to the `simple_set` interface. (Standard library sets are implemented as balanced search trees.) *This set is for illustration purposes only; the*

code you write is not to make use of any of the collections in the standard library.

Building on `simple_set`, the provided code then defines a more complex `range_set`:

```
template<class T, class C = comp<T>>
class range_set : public virtual simple_set<T> {
  public:
    virtual range_set<T, C>& operator+=(range<T, C> r) = 0;
    virtual range_set<T, C>& operator-=(range<T, C> r) = 0;
};
```

where `range` is defined as follows

```
class empty_range { };      // exception
template<typename T, typename C = comp<T>>
class range {
    T L;        // represents all elements from L
    bool Linc;  // inclusive?
    T H;        // through H
    bool Hinc;  // inclusive?
    C cmp;
    static const empty_range err;
  public:
    range(const T l, const bool linc, const T h, const bool
hinc)
            : L(l), Linc(linc), H(h), Hinc(hinc), cmp() {
        if (cmp.precedes(h, l)
            || (cmp.equals(l, h) && (!Linc || !Hinc)))
throw err;
    }
    bool contains(const T& item) const {
        return ((cmp.precedes(L, item) || (Linc &&
cmp.equals(L, item)))
            && (cmp.precedes(item, H) || (Hinc &&
cmp.equals(item, H))));
    }
};
```

Here `C`'s `precedes` and `equals` methods are used to determine whether a given element lies within the range. The default comparator `comp`

employs `T`'s `operator<`, assuming it exists. Again, you might want a different comparator for certain types.

So what is a `range_set` good for? It allows you to insert and remove contiguous ranges of elements from the set *en masse* and, *in some implementations*, in asymptotically less time than it would take to remove the individual elements.

The provided code defines an `std_range_set`, again as an example, but this implementation does *not* have good performance. The problem is that the sets of the standard library are defined to hold individual elements only; they do not capture ranges.

You are to implement three versions each of `simple_set` and `range_set`:

`carray_simple_set<T>` and `carray_range_set<T, C, I>`
Characteristic arrays. Constructor takes arguments `l` and `h` of type `T`, which must be coercible to `int`; insert and remove routines throw an `out_of_bounds` exception if passed an element outside the half-open range [`l`, `h`). Insertion, removal, and lookup of individual elements must all take constant time. For `carray_range_set`, `T` must support an increment operation, and insertion and removal of ranges can take time linear in the size of the range. Must be space efficient: `carray_simple_set(l, h)` must consume `h` − `l` + c bits, for some small constant c.

`hashed_simple_set<T, F>` and `hashed_range_set<T, F, C, I>`
Constructor takes a single argument of type `int`, specifying the maximum number of elements that can belong to the set at any one time. Insert routine throws an `overflow` exception if there is no more room in the table. Optional template parameter `F` specifies a function-object class that can be used to convert a `T` object into an int. Insertion, removal, and lookup of individual elements should take expected constant time. For `hashed_range_set`, `T` must support an increment operation, and insertion and removal of ranges can take time linear in the size of the range.

`bin_search_simple_set<T, C>` and `bin_search_range_set<T, C>`

Constructor takes a single argument of type `int`, specifying the maximum number of elements (or, for `bin_search_range_set`, ranges) that can belong to the set at any one time. Insert routine throws an `overflow` exception if there is no more room. Insertion and removal of individual elements or ranges may take time linear in the number of elements or ranges currently in the set, worst case, but must take amortized constant time if elements are inserted in sorted order. Lookup must take logarithmic time.

For `carray_range_set` and `hashed_range_set`, optional template parameter `I` specifies a function-object class that can be used to increment a `T` object, to iterate over the elements of a range. `carray_range_set` should inherit from `range_set` and `carray_simple_set`; `hashed_range_set` and `bin_search_range_set` should similarly inherit from both `range_set` and their corresponding non-range version. See the `std_range_set` for an example of how to do this.

Because its internal data structure represents individual elements only, the `std_range_set`, like `carray_range_set` and `hashed_range_set`, can be instantiated only for types that support an increment operation. As noted above, this means that ranges can't be inserted and removed efficiently. It also means that the set cannot support ranges containing an unbounded number of elements. Ideally, one would like to be able to create a set of real numbers, and specify that it contains not only 1.2, 4.67, and –0.235, but also all values from 123 to 158.5, possibly including either or both endpoints. Your `bin_search_range_set` must allow this. When instantiated for strings, it must also allow ranges like ["apple", "orange"), which includes an unbounded number of strings.

Warning: keep in mind that ranges have the potential to overlap, both with individual elements and with each other. If you insert a range that overlaps an existing range, you'll need to merge them. If you remove a range from the middle of an existing range, you'll need to split it. A particularly tricky case arises when you remove an individual element from the middle of a range. If the range isn't finite, the two halves of the

resulting split will need to be open on the side next to the removed element.

Division of labor and writeup

As in previous assignments, you may work alone or in teams of two. If you choose to work in pairs, the obvious division of labor is to divide up the implementations. If you do this, take care to figure out what common assumptions you need to share with your partner. Each two-person team need only turn in a single `README`; make sure it includes the names of both partners.

Be sure to follow all the rules on the Grading page. As with all assignments, use the turn-in script: `~cs254/bin/TURN_IN`. Put your write-up in a `README.txt` or `README.pdf` file in the directory in which you run the script. Be sure to describe any features of your code that the TAs might not immediately notice.

See the Resources page for help with C++.

Extra Credit suggestions

1. Add standard library-style iterators to the sets for which they make sense.
2. Implement `operator=`, `operator==`, `operator!=`, `operator|` (union), `operator&` (intersection), and `operator-` (difference). Can you make these work for sets with the same element type but different implementations? (E.g., can you compare a `carray_simple_set<T>` and a `hashed_simple_set<T>`?) Can you make them work for `range_sets` whose ranges aren't finite?
3. Measure the performance of operations on your different set implementations and discuss the results.
4. Implement a balanced search tree with efficient (not necessarily finite) ranges—e.g., of strings or doubles.
5. Adapt your code to implement dictionaries (maps) instead of sets.
6. Try the assignment in Java, C#, or Ada.
7. Implement an *adaptive*, intelligent set that keeps internal statistics and changes its implementation based on the distribution of elements.

8. Make your code thread-safe (properly synchronized), so all the operations will appear to occur atomically if they are employed in a multithreaded program.

NB: One could design both the set abstraction and the concrete implementations in many different ways. The starter code I've given you may not suit your fancy. Feel free to explore other APIs for extra credit, but if you do, put them in separate files, and be sure to respect the given API for the main assignment; otherwise the TAs won't be able to test your code with their driver scripts and you won't get the credit you want.

Trivia Assignment

Before the beginning of class on **Wedmesday, December 4**, *each student* should complete the T6 trivia assignment found on Blackboard.

MAIN DUE DATE: 11:59 pm, **Friday December 13**; no extensions.