CS246E—Assignment 2 (Fall 2021)

B. Lushman

Due Date 1: Monday, October 18, 5pm
Due Date 2: Friday, October 29, 5pm

Part of this assignment is due on Due Date 1; the rest is due on Due Date 2. See the assignment questions for specifics. All code is to be written in C++14.

1. Sometimes we want to iterate a data structure from back to front. For this purpose, we could imagine a *reverse iterator*, which supports the same operations as an ordinary iterator, except that those operations navigate the data structure in the opposite direction. To create a reverse iterator, we call the method `rbegin`, which creates an iterator pointing at the last item in the collection. The method `rend` produces an iterator that can be used as a stopping condition for reverse iteration, and `operator++` mutates the iterator so that it now points one item *before* where it previously pointed.

   Implement reverse iteration, and the methods `rbegin` and `rend`, for the `vector` class. Be careful, though, about undefined behaviour! The standard permits a pointer to point one item past the end of an array (so long as that pointer is never dereferenced); it does not permit a pointer to point one item before the beginning of an array. Also remember that your solution will need to work for both constant and non-constant vectors.

   Use the `vector` implementation in `lectures/p9` as your starting point. Add your code directly to the vector class. **Submit `vector.h` and `vector.cc`. Do not submit `main.cc`.**

   No test suite required; submit your code on Due Date 1.

   Please note that correctness marks awarded by Marmoset constitute only one part of your overall grade on this question. Your code will be carefully hand-marked to ensure that it is implemented correctly and properly designed. You must, therefore, make sure your code is very clearly readable. If the TA is unable to follow your solution, it will be assumed to be wrong.

2. Implement the insert method for the `list` class from lectures:

   ```
   iterator insert(iterator posn, int x); // inserts x at position posn
   ```

   Keep in mind that the `list` class from lectures is singly-linked, not doubly-linked, and you are not permitted to change this. Specifically, you may not change the class `list::Node`. You may make changes to the other classes, but you may not change the interface of class `list` (other than to add the `insert` method), and if you choose to add or change fields, you may only change the sizes of your classes by a constant amount of space (e.g., you may not replicate or simulate the list in another data structure).

   Use the `list` implementation in `lectures/p9` as your starting point. Add your code directly to the list class. **Submit `list.h` and `list.cc`. Do not submit `main.cc`.**

   No test suite required; submit your code on Due Date 2.

   Please note that correctness marks awarded by Marmoset constitute only one part of your overall grade on this question. Your code will be carefully hand-marked to ensure that it is implemented correctly and properly designed. You

must, therefore, make sure your code is very clearly readable. If the TA is unable to follow your solution, it will be assumed to be wrong.

3. A *deque* (i.e., "double-ended queue") is a linear data structure that supports efficient insertion and removal from both ends, without shuffling of the items in the deque (recall that a vector can avoid shuffling only at one end). Moreover, a deque guarantees that when it needs to grow, there is no reallocation and copying (or moving) of the items (of type `T`) in the deque. There may, however, be reallocation and copying of pointers.

   The typical way of structuring a deque (and the way you are to do it) is as an array of pointers, where each pointer points to a structure containing a fixed-length (for our purposes, let's say 10), partially-filled array of `T` objects. The arrays of objects, taken in sequence, should all be completely full of `T` objects, except for the first one (which may have some empty slots at the front) and the last one (which may have some empty slots at the back). The array of pointers should have some emtpy slots set aside on both ends, so that new structures can be added to the front or back, as needed. If you run out of slots in this array, you can reallocate this array and copy the pointers, without touching the actual data items, and this reallocation will run in amortized constant time, without any copies or moves on `T` objects (which may be considerably more expensive than copying pointers).

   Moreover, although the entire deque is not stored in contiguous memory, large chunks of it are, which provides good locality and cache performance.

   All classes that you build in support of your solution must reside in the `cs246e` namespace. Your map must support a default constructor, an initializer list constructor, the big 5 (or 4), and the following methods:

   ```
   operator[]  at  begin  end  empty  size
   push_front  pop_front  push_back  pop_back
   ```

   Semantics can be found at `cplusplus.com` (search for `deque`).

   We will provide a test harness, suitable for testing your code. After each command you issue to the test harness, it will print all deques that it knows of, using a range-based loop (so you need to have `begin` and `end` working; until you have finished them, you can replace this in the harness with your own print routine; just remember that our test harness will use a range loop).

   You will need to submit a test suite for your program, in a format compatible with `runSuite`. For simplicity, you may limit your testing to just the constructors, `operator=`, `push/pop_front/back`, and `operator[]`.

   Submit your test suite on Due Date 1; submit your code on Due Date 2.

   Submit `deque.h`

   Please note that correctness marks awarded by Marmoset constitute only one part of your overall grade on this question. Your code will be carefully hand-marked to ensure that it is implemented correctly and properly designed. You must, therefore, make sure your code is very clearly readable. If the TA is unable to follow your solution, it will be assumed to be wrong.

4. A *map* (also known as a table or dictionary) is a data structure that provides key-value lookup. The central operation of a map is the lookup operation that, given a key, produces the associated value. Keys are assumed to be unique within the map; values need not be. In this problem, you will implement the `map` template.

   Your map must use a binary search tree as its underlying data structure. Although balance is necessary for efficient searches, we do not require for this exercise that you maintain balance in your tree.

All classes that you build in support of your solution must reside in the `cs246e` namespace. Your map must support a default constructor, an initializer list constructor, the big 5 (or 4), and the following methods:

```
operator[]  at  begin  end  empty  size  erase(const Key &k)  clear  count
```

Semantics can be found at `cplusplus.com` (search for `map`). Note that if `operator[]` is called using a key that is not present, a new node is created for that key, with the value default-constructed.

We will provide a test harness, suitable for testing your code. After each command you issue to the test harness, it will print all maps that it knows of, using a range-based loop (so you need to have `begin` and `end` working; until you have finished them, you can replace this in the harness with your own print routine; just remember that our test harness will use a range loop).

You will need to submit a test suite for your program, in a format compatible with `runSuite`. For simplicity, you may limit your testing to just the constructors, `operator=`, `operator[]`, `erase`, and `clear`.

The running time complexity of `operator++` for your iterator class is permitted to be linear in the path length from the current node in the map to the next node.

Submit your test suite on Due Date 1; submit your code on Due Date 2.

Submit `map.h`

Please note that correctness marks awarded by Marmoset constitute only one part of your overall grade on this question. Your code will be carefully hand-marked to ensure that it is implemented correctly and properly designed. You must, therefore, make sure your code is very clearly readable. If the TA is unable to follow your solution, it will be assumed to be wrong.