Informatik - AS19

Exercise 12: Iterators and Containers

Handout: 2. Dez. 2019 06:00

Due: 9. Dez. 2019 18:00

Task 1: Lexicographic comparison

Open Task

Task

Words in, e.g., a dictionary or an index are sorted lexicographically as given by the alphabet.

Task: Implement a lexicographic comparison function for strings using the alphabetical ordering provided by the ASCII code.

A string a is lexicographically smaller than a string b, if

- the first character of a in that both a and b differ is smaller than the corresponding character of b (e.g., `bicycle < bike` because the first different characters are `c < k`).
- the string a forms the start of b, but is shorter (e.g., `web < website`).

1. Write a function that compares two strings. The function must return true if the first string is smaller than the second with respect to *lexicographic order*. The function must, using `using Iterator = std::string::iterator;` have the following signature:

```
// PRE: [first1,last1) and [first2,last2) are valid ranges
// POST: returns true if string at [first1,last1) is lexicog
//       smaller than string at [first2,last2)
bool lexicographic_compare(Iterator first1, Iterator last1,
                           Iterator first2, Iterator last2)
```

2. Write a program using this function to compare two strings given as standard input, and output the lexicographic minimum of the two. If the two strings are identical, the program should instead print `EQUAL`.

Note: Using `string` comparison functions from the standard library is not allowed. Note that the == operator on strings is implemented by `std::string`, and thus counts as a library function.

In the template code, we provide function `print` which demonstrates how to print a string (or part of it) using iterators.

Input

Two words, separated by whitespace.

Example:

```
bike bicycle
```

Output

The minimum of the two input strings, with respect to the lexicographical order, or `EQUAL` if both strings are equal.

Example:

```
bicycle
```

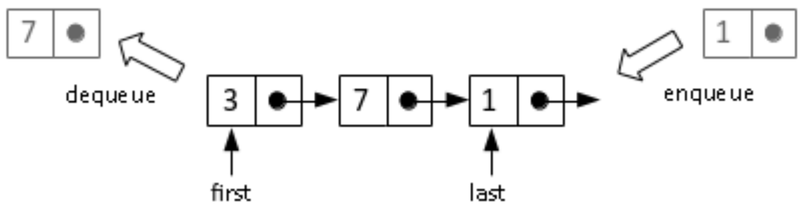
Task 2: Dynamic Queue

Open Task

Task

Your objective is to implement your own queue class with integer values as a dynamic data structure.

A queue provides the two basic operations: enqueue and dequeue. The operation *enqueue* adds a new element to the back of the queue. The operation *dequeue* removes the first element from the queue:



A common way to implement a queue is using a linked list, as the one that was shown in the lecture. In order to be able to both enqueue and dequeue, we need to access respectively the first and last elements of the list.

Procedure: The queue declaration is already provided. Your task is to fill the missing definitions for the required functions, marked by comments *// TODO* in `queue.cpp`. Function annotations (pre- and postconditions), found along declarations in the header `queue.h`, explain what each function is supposed to do. You may (and probably will have to) add auxiliary non-member functions in `queue.cpp` in order to implement some of the functions.

A class invariant helps to detect implementation problems early. For this queue, we have the following class invariant: Either both pointers `first` and `last` are set to `nullptr` or both are not set to `nullptr`. Check the invariant at suitable places, e.g., after modifications of the queue data members.

Subtasks:

1. Implement the default constructor of class `Queue`.
2. Implement private member function `is_valid`, which should assert the class invariant.
3. Implement the queue operations: member functions `enqueue`, `dequeue` and `is_empty`.
4. Implement the member function `print_reverse` to print the content of a queue to an output stream in reverse order. To output the queue content (output operator), use the following format: bracket open (`[`) , zero or more integer numbers separated by spaces, bracket close (`]`). E.g., the empty queue must be output like this: `[]`, the queue containing elements 13, 7, and 42 (from first to last): `[42 7 13]`.

   The function `print_reverse` **must** be implemented through an auxiliary function traversing the queue recursively, from a `Node` pointer given as argument. In particular, the implementation of `print_reverse` **must not** use loop structures. We provide implementation of function `print` printing the content of a queue in regular order as a reference.
5. *Optional:* Think of ways to implement `print` and `print_reverse` with loops instead of recursion (do **not** change your submission). In particular, try to find a way to implement `print_reverse` that does not make use of unbounded auxiliary storage.

Testing: The test input consists of a list of function calls in text representation. The template includes the parser for this text representation to avoid a lengthy specification. You do not have to implement it, but you may want to take a look at it to understand how it works. Also you can use it to do test your queue manually.

The `main` function creates a queue that is initially empty, and on which member functions are later called. The following EBNF defines the input:

```
queue_ops = queue_op { queue_op } "end" .
queue_op  = enqueue | is_empty | dequeue | print | print_revers

// enqueues element
enqueue = "enqueue" integer .

// returns whether queue is empty
is_empty = "is_empty" .

// dequeues element
dequeue = "dequeue" .

// prints queue contents in reverse
print_reverse = "print_reverse" .

integer  = C++ integer value
```

Task 3: Decomposing a Set into Intervals

Open Task

Task

Any finite set of integers can be uniquely decomposed as a union of disjoint maximal integer intervals, e.g as the union of non-overlapping intervals which are as large as possible. For example, the set $X = \{1, 2, 3, 5, 6, 8\}$ decomposes as $X = [1, 3] \cup [5, 6] \cup [8, 8]$. Note that $X = [1, 2] \cup [3, 3] \cup [5, 6] \cup [8, 8]$ or $X = [1, 3] \cup [5, 6] \cup [5, 6] \cup [8, 8]$ are not valid decompositions, as $[1, 2]$ and $[3, 3]$ are not maximal interval subsets of X, and intervals may not repeat.

Write a program that inputs a set of integers, as the (possibly repeating) sequence of its members, and outputs the interval decomposition of the set in ascending order of boundaries.

Reminder: ordered sets can be represented in C++ using the `std::set<...>` container, in which elements are added using the `insert` method. Iteration (using iterators) in ordered sets takes place in increasing order.

Hint: You may first define a function that, from two `std::set` iterators `first` and `last`, finds the largest integer interval starting from `first` and ending before `last`, and returns an iterator just after this interval.

Input

A set of non-negative integer, given as the sequence of its members followed by a negative integer. The sequence can be in any order and may feature repetitions.

Example:

```
3 5 8 6 5 3 2 1 -1
```

Output

The interval decomposition of the input set, with interval given in increasing order of boundaries. The interval decomposition must be given as a parenthesized sequence of intervals, with intervals separated by the character `U`. Intervals themselves must be given by their lower and upper bound, separated by a comma and parenthesized by brackets, with the lower bound coming first.

Example:

```
([1,3]U[5,6]U[8,8])
```